**Keycloak Integration**

**🔐 1. What is Keycloak?**

Keycloak is an open-source Identity and Access Management (IAM) solution developed by Red Hat. It provides authentication, authorization, and user management features for modern applications — both web and mobile.

With Keycloak, developers can offload all security-related tasks, like:

* User login and logout
* Single Sign-On (SSO)
* Role-based access control (RBAC)
* Multi-factor authentication (MFA)
* Identity brokering (social logins like Google, Facebook)
* Fine-grained permission control

Keycloak follows OAuth 2.0 and OpenID Connect (OIDC) protocols for secure token-based access across systems.

**2. Core Concepts in Keycloak**

**2.1 Realm**

A **Realm** in Keycloak is a security domain that manages a set of users, credentials, roles, and clients. Realms are completely isolated from one another, allowing multiple tenants or environments (e.g., dev, staging, production) to exist on a single Keycloak server without interference.

**2.2 Clients**

A **Client** represents an application or service that communicates with Keycloak to authenticate users or access protected resources. Clients can be:

* **Public clients** (e.g., frontend SPAs like Angular) that do not use a secret.
* **Confidential clients** (e.g., backend APIs) that authenticate using a client secret or certificate.

**2.3 Users**

**Users** are entities that represent individuals who can authenticate and interact with protected applications. Users have credentials (e.g., username, password) and can be assigned roles and groups for access control.

**2.4 Roles**

**Roles** are permissions or access levels that can be assigned to users, groups, or clients. Roles can be:

* **Realm roles**: Global across the realm.
* **Client roles**: Specific to a particular client/application.

**2.5 Groups**

**Groups** are a way to organize users into logical collections. Roles can be assigned at the group level, and users inherit those roles by being part of the group.

**2.6 Scopes**

A **Scope** defines the level of access that a client is requesting. Scopes often represent permissions or resource access levels (e.g., read, write).

**2.7 Client Scopes**

**Client Scopes** are reusable definitions of scope and protocol mappers. They allow administrators to define common sets of token claims that multiple clients can share, improving configuration consistency.

**2.8 Protocol Mappers**

**Protocol Mappers** are used to define which user information (claims) is included in tokens. This includes data such as usernames, email addresses, and user roles. Mappers are essential for customizing access tokens and ID tokens.

**2.9 Access Token**

An **Access Token** is a short-lived JSON Web Token (JWT) issued by Keycloak. It contains claims that authorize a client or user to access protected resources. It is typically sent in the Authorization header as a Bearer token.

**2.10 ID Token**

An **ID Token** is a JWT that contains information about the authenticated user, such as username, email, and other identity attributes. It is primarily used by client applications (like frontend apps) to personalize the user interface.

**2.11 Refresh Token**

A **Refresh Token** is a long-lived token that allows clients to obtain new access tokens without requiring the user to re-authenticate. This is particularly useful for maintaining user sessions in long-lived applications.

**2.12 Token Exchange**

**Token Exchange** is a feature in Keycloak that enables one token to be exchanged for another, often with a different audience or set of scopes. This is commonly used for service-to-service communication, where a backend service needs to act on behalf of the user or itself when calling another service.

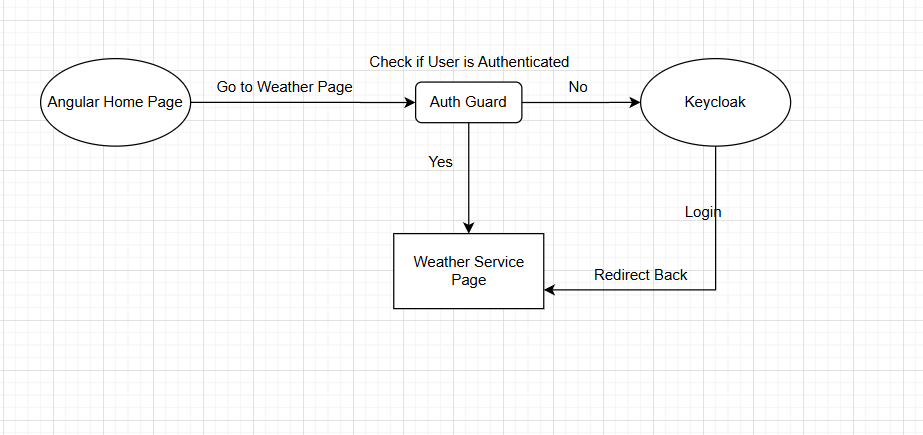
**3. Authentication Flow Overview**

**3.1. Angular → Keycloak Login → Access API**

* When the Angular user navigates to WeatherNow, the **Auth Guard** checks authentication.
* If not authenticated, the user is redirected to **Keycloak's login page**.
* Upon login, Keycloak returns an **ID Token** and **Access Token**.
* These tokens contain the **audience** for the allowed APIs (TestApi, ServiceB) via client scope mappers.
* After login, the user is redirected back to Angular and can now access the APIs using the access token.

**3.2. Angular Auth Config Highlights**

* Uses keycloak-js package.
* Client configured in Keycloak with proper **redirect URIs**, **client scopes**, and **roles**.
* AuthGuard checks keycloak.isAuthenticated() if needed.



**4. Service-to-Service Communication in Backend**

**ServiceB** initiates a secure backend-to-backend call to TestApi. This communication is handled using the **Client Credentials Flow**, a common OAuth 2.0 flow for service accounts.

This does **not** involve user tokens or token exchange. Instead, **ServiceB** acts on its own behalf using its **client ID** and **client secret** to obtain an access token from Keycloak.

**4.1 Flow Description**

1. **ServiceB** wants to securely call an endpoint on **TestApi**.
2. **ServiceB** is configured in Keycloak as a **confidential client** with **Service Accounts Enabled**.
3. It authenticates itself using its **client ID** and **client secret** to request an **access token** from Keycloak using the **Client Credentials Grant**.
4. Keycloak issues a short-lived **access token** with claims based on the roles or scopes assigned to ServiceB.
5. **ServiceB** adds the access token to the Authorization header as a Bearer token when calling **TestApi**.
6. **TestApi** receives the request and validates the access token to ensure it is:
   * Issued by Keycloak (signature verification)
   * Intended for a valid audience
   * Possibly has required scopes or roles (optional)

**4.2 Why This Matters**

This approach establishes:

* **Stateless, secure communication** between services
* **No user context** — purely machine-to-machine
* Access control based on the identity of **ServiceB**
* Token configuration and access rights managed centrally in Keycloak

