#Author Sandro Lipinski
#Created May 2025
#Introduction Basic Keycloak setup with Spring Boot as Resource Server and Angular Application using OAuth 2.0, Authorization Code Flow and some References.

https://docs.spring.io/spring-security/reference/servlet/oauth2/index.html#oauth2-resource-server
https://www.keycloak.org/docs/latest/server_admin/#_security-hardening
https://www.keycloak.org/docs/latest/authorization_services/index.html#_overview

OAuth 2.0 Security Best Practices

Endpoints:
Keycloak will expose all relevant URI's at:
/realms/{realm-name}/.well-known/openid-configuration

Authentication & Authorization

Authentication
"Who are you?"
Uses OpenID Connect (OICD).

Authorization
"What are you allowed to do?"
Uses OAuth 2.0

OpenID Connect

Authentifizerungsschicht, das auf OAuth 2.0 basiert. Ermöglicht es Anwendungen (Clients) die Identität eines Nutzers zu überprüfen.

OAuth 2.0

Protokoll zur sicheren Autorisierung für Desktop-, Web- und Mobile-Anwendungen. Ermöglicht es einem Nutzer (User/ Resource Owner), einer Anwendung (Clients/Third Parties) den Zugriff auf seine Daten, ohne seine geheimen Anmeldeinformationen preiszugeben.

Authentication Flows/Authorization Grants

Which OAuth 2.0 Flow Should I Use?

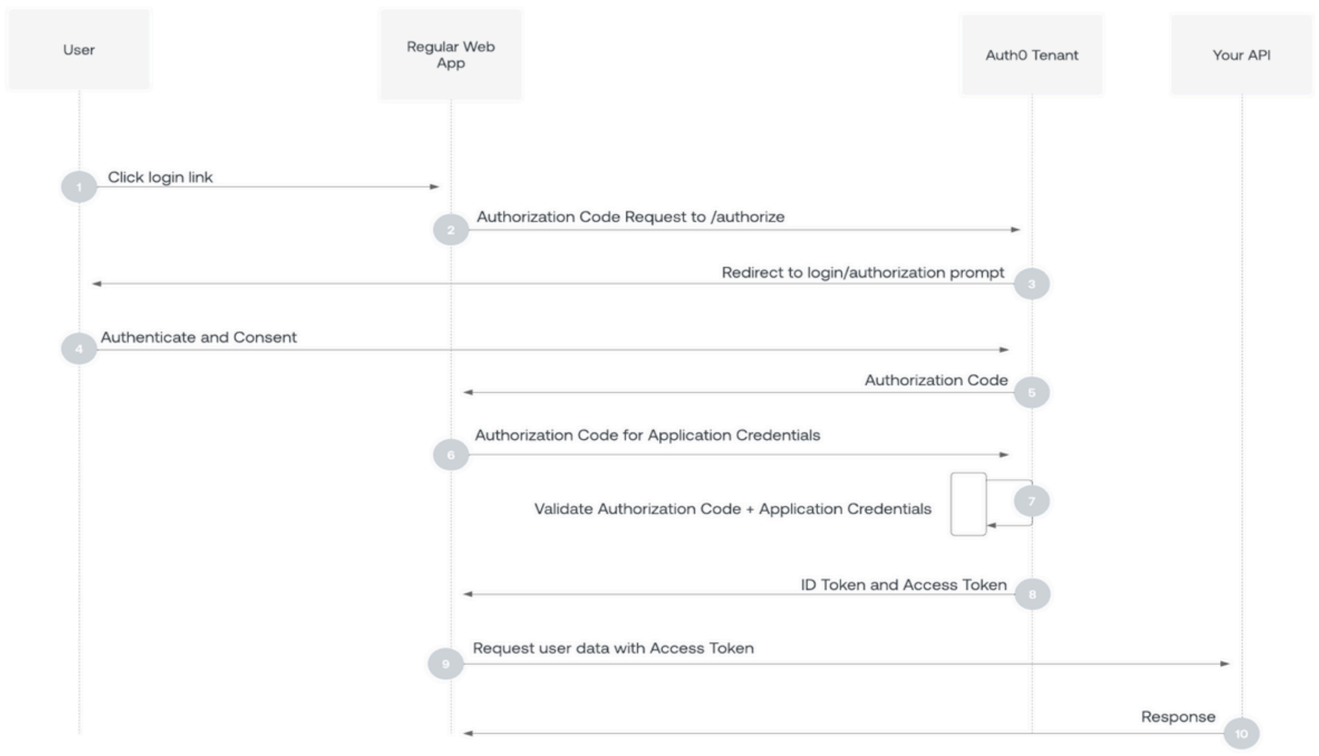
OAuth uses term "Grant", Open ID Connect uses term "flow". Therefore some sources will refer to both grant and flow, e.g. "Authorization code grant flow" (ebay).

Authorization Code Grant (Keycloak: Standard flow)

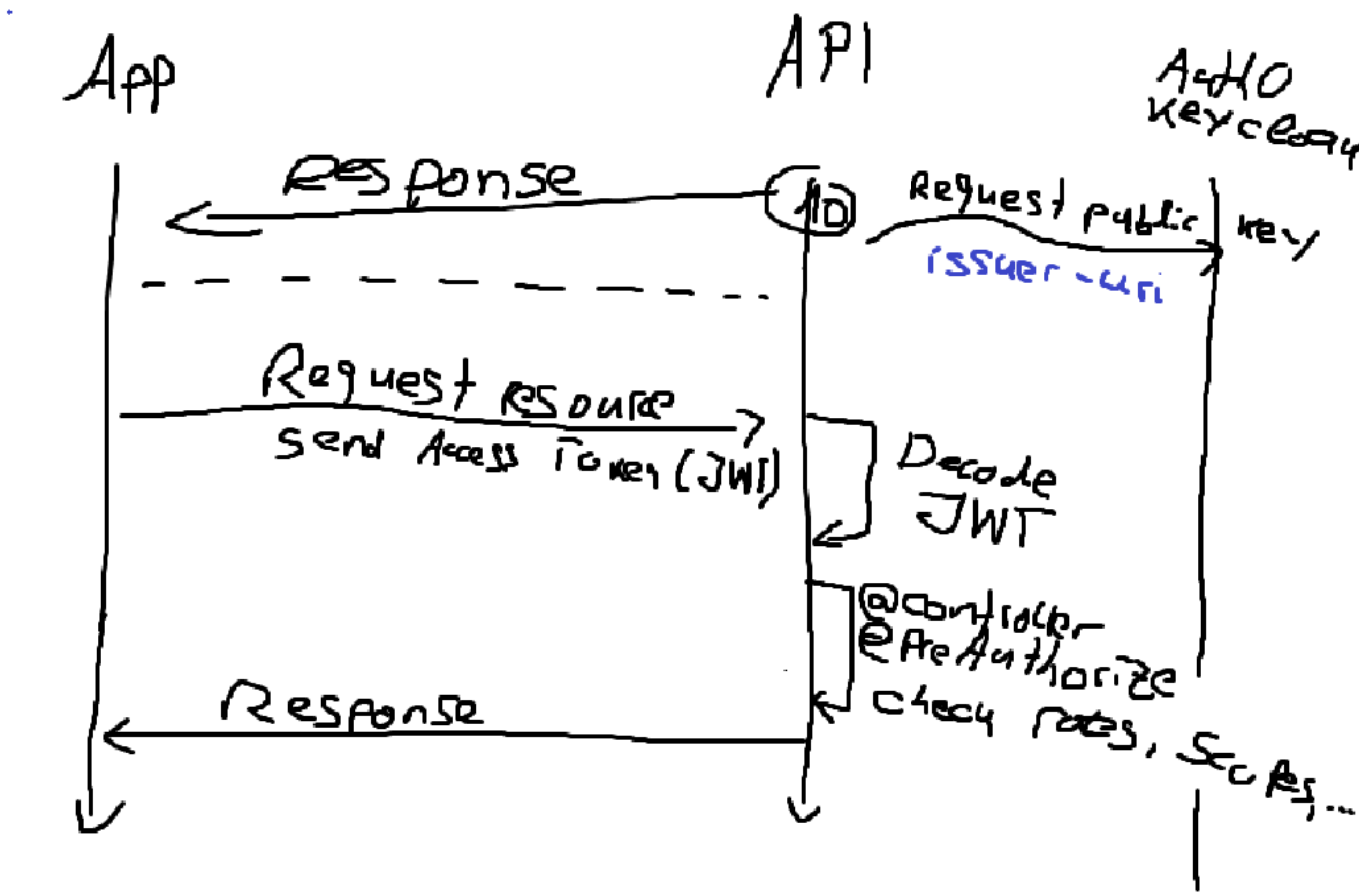
For browser based applications like Angular. Enables standard OpenID Connect redirect based authentication with authorization code.

Recommended approach for web applications.

grant_type=authorization_code


User
Regular Web App
Auth0 Tenant
Your API
1 Click login link
2 Authorization Code Request to /authorize
3 Redirect to login/authorization prompt
4 Authenticate and Consent
5 Authorization Code
6 Authorization Code for Application Credentials
7 Validate Authorization Code + Application Credentials
8 ID Token and Access Token
9 Request user data with Access Token
10 Response

Client requesting resources, after authentication & authorization:



1. A user connects to an application using a browser. The application detects the user is not logged into the application.
2. The application redirects the browser to Keycloak for authentication.
3. The application passes a callback URL as a query parameter in the browser redirect. Keycloak uses the parameter upon successful authentication.
4. Keycloak authenticates the user and creates a one-time, short-lived, temporary code.
5. Keycloak redirects to the application using the callback URL and adds the temporary code as a query parameter in the callback URL.
6. The application extracts the temporary code and makes a background REST invocation to Keycloak to exchange the code for an *identity* and *access* and *refresh* token. To prevent replay attacks, the temporary code cannot be used more than once.

Implicit Flow (Password Grant)
LEGACY!
For browser based applications, less secure then Authorization Code Flow, not recommended and deprecated in OAuth 2.1.

1. A user connects to an application using a browser. The application detects the user is not logged into the application.

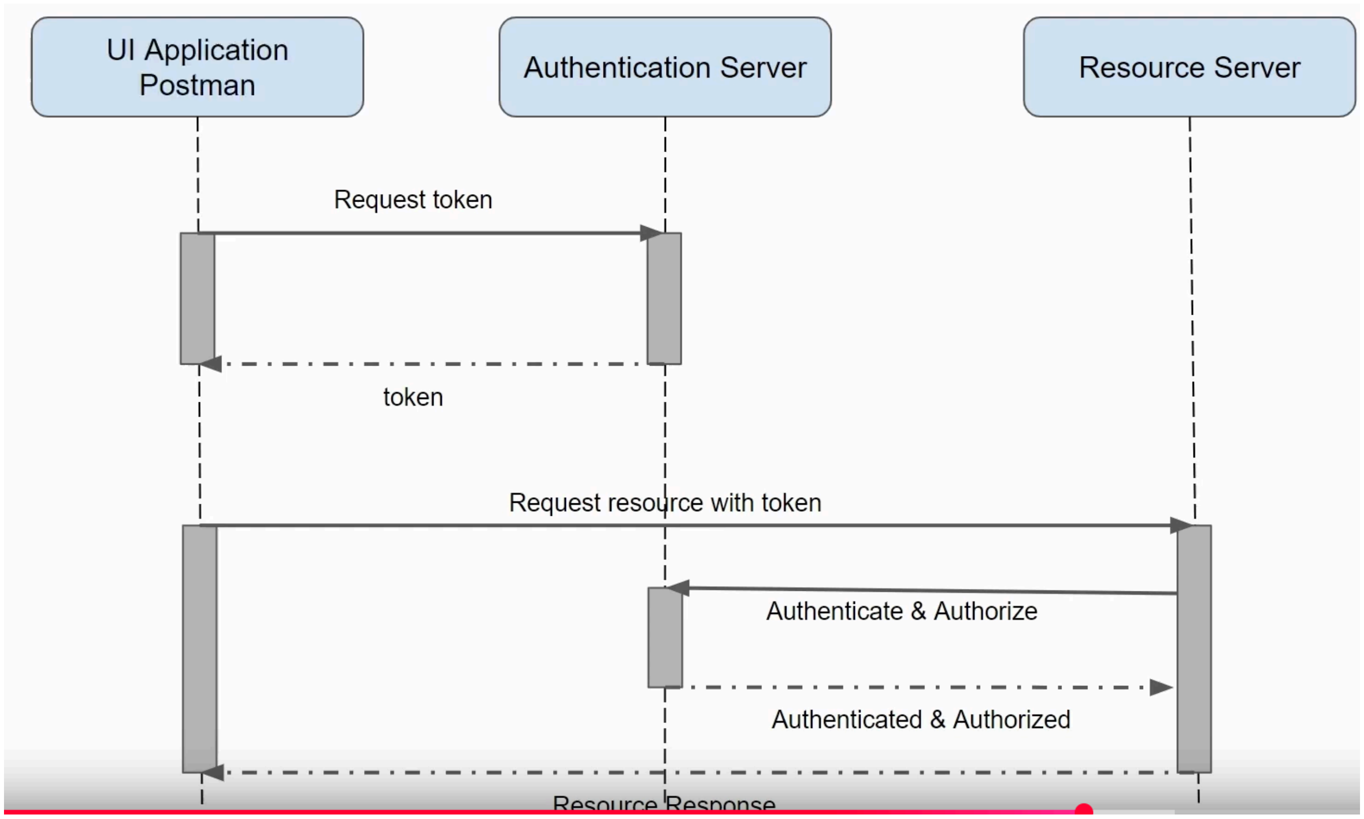
2. The application redirects the browser to Keycloak for authentication.
3. The application passes a callback URL as a query parameter in the browser redirect. Keycloak uses the query parameter upon successful authentication.
4. Keycloak authenticates the user and creates an *identity* and *access* token. Keycloak redirects to the application using the callback URL and additionally adds the *identity* and *access* tokens as a query parameter in the callback URL.
5. The application extracts the *identity* and *access* tokens from the callback URL.

Resource Owner Password Credentials Grant (Keycloak: Direct access grants)
LEGACY! Testing purposes only.
For REST applications which can not support modern authentication protocols. Involves sharing credentials with another service, caution!
OAuth was exactly invented to prevent this.

grant_type=password

Client Credentials Grant (Keycloak: Service account roles)
For REST applications to request an access token to their own resources, not on behalf of a user.
If Client == Resource Owner.

grant_type=client_credentials



OIDC Logout

Session Management:
Browser-based logout. Application obtains session status information from Keycloak at regular basis. If session gets terminated at Keycloak side, the application will notice and trigger its own logout.

RP-Initiated Logout:
Browser-based logout. This usually happens when the user clicks the "Logout" Button at the frontend application. This will redirect the user to a specific endpoint at Keycloak. Keycloak will then tell the clients to invalidate their tokens and enventually redirect the client back to the application, when the process is finished.

Client will get redirected to `post_logout_redirect_uri`, needs to match one of the `Valid Post Logout Redirect URIs` specified in the client configuration.

Front-channel Logout:

Back-channel Logout:

JSON Web Token (JWT)

Used as ID Token (OpenID Connect) & Access Token

Traditional you would use sessions, where a cookie was stored on the client and validated on the server. The server would then store the session id's, so the authentication state would be handled on the server side. This introduced a bottleneck, since you can hardly scale out your server infrastructure horizontally, if all the servers have to store that token.

JWT solves that problem, with the client sending its login details to the server, with the server creating a JWT with a private key and then sent back to the browser, where it is kept. The browser will then send that Token with its request, prefixxed by "Bearer". The server only has to validate that signature and does not need to store the sessions anymore. This is a lot more efficient for distributed systems, since the server does not have to do database lookups somewhere else on the infrastructure.

BUT, JWT's still can be hijacked by an attacker, are hard to invalidate and can not be used to authenticate a user on the backend.

Shorthand: With cookies the session is managed server-side, with tokens on the client side.

See how your JWT token looks like: https://jwt.io/.
Be careful, DO NOT use your real tokens, only use dummy tokens that will not get used anymore.

Example JWT, base64 encoded:

```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI1Vk9jb0JuMmNBMkFmdTZvT0JxVV
dseHVLSnNuUkJoT0h4M2dHc0h4S053In0.eyJleHAiOjE3NDY4ODM3NTYsImlhdCI6MTc0Njg4MzQ1
NiwianRpIjoib25ydHJvOjNlMTMxOGVlLTgwNWUtNDViZC04MWRmLWZiNjVjZmJlNTE2MiIsImlzcy
I6Imh0dHBzOi8vYXV0aC5pbnN5LmhzLWVzc2xpbmdlbi5jb20vcmVhbG1zL2luc3kiLCJhdWQiOiJh
Y2NvdW50Iiwic3ViIjoiOGI5OTcyNzktOWI3NS00MTg1LTk1NDktY2IyOGE1YTljNGEyIiwidHlwIj
oiQmVhcmVyIiwiYXpwIjoic3ByaW5nLWJvb3QtcmVzdC1jbGllbnQiLCJzaWQiOiI2YTg1NTRiNC03
M2Q1LTQ4OGMtOTMwNi05YmFjZWRmOTg2NzAiLCJhY3IiOiIxIiwiYWxsb3dlZC1vcmlnaW5zIjpbIi
oiXSwicmVhbG1fYWNjZXNzIjp7InJvbGVzIjpbImRlZmF1bHQtcm9sZXMtaW5zeSIsIm9mZmxpbmVf
YWNjZXNzIiwidW1hX2F1dGhvcml6YXRpb24iXX0sInJlc291cmNlX2FjY2VzcyI6eyJhY2NvdW50Ij
p7InJvbGVzIjpbIm1hbmFnZS1hY2NvdW50IiwibWFuYWdlLWFjY291bnQtbGlua3MiLCJ2aWV3LXBy
b2ZpbGUiXX19LCJzY29wZSI6InByb2ZpbGUgZW1haWwiLCJlbWFpbF92ZXJpZmllZCI6ZmFsc2UsIm
5hbWUiOiJtYXggbXN1dGVybWFubiIsInByZWZlcnJlZF91c2VybmFtZSI6InNhbGlpdDAxQGhzLWVz
c2xpbmdlbi5kZSIsImdpdmVuX25hbWUiOiJtYXgiLCJmYW1pbHlfbmFtZSI6Im1zdXRlcm1hbm4iLC
JlbWFpbCI6InNhbGlpdDAxQGhzLWVzc2xpbmdlbi5kZSJ9.b7-yv6SWXJV9tA-
K4G61f3iiixumOtG8_FrZnZxdwDwYZBmwTQWJxvHgRE-
nFZWhUrot_vGa6BlmQlFZioD8UgA3PvZOnVIOa-
JDWg453EkqGZREAxHPwq7jYKNNr7HNd8YHj39W38tIONJ51O99uZoFZtS86VDBguQ0MhMIPNanG2c-
T18q5pgZUCBLMVHoa5YXXPV7_yNWOxLLhZOCVwI7M6kE-
4TluieQxd_eq1ZLVxwWp_1GjaUUUDOwb8k-
cht4udcmMPX6zQmWaINAQKprbj2lhZNMGAjUErEXtfnrL3KmufdULVfNU1allHIeX8ej4z9drPHg2h
w3ffdq_g
```

Decoded JWT:

```
Header:
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "5VOcoBn2cA2Afu6oOBqUWlxuKJsnRBhOHx3gGsHxKNw"
}

Payload:
{
  "exp": 1746883756,
  "iat": 1746883456,
  "jti": "onrtro:3e1318ee-805e-45bd-81df-fb65cfbe5162",
  "iss": "https://auth.insy.hs-esslingen.com/realms/insy",
  "aud": "account",
  "sub": "8b997279-9b75-4185-9549-cb28a5a9c4a2",
  "typ": "Bearer",
  "azp": "spring-boot-rest-client",
  "sid": "6a8554b4-73d5-488c-9306-9bacedf98670",
  "acr": "1",
  "allowed-origins": [
    "*"
  ],
  "realm_access": {
    "roles": [
      "default-roles-insy",
      "offline_access",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "profile email",
  "email_verified": false,
  "name": "max msutermann",
  "preferred_username": "saliit01@hs-esslingen.de",
  "given_name": "max",
  "family_name": "msutermann",
  "email": "saliit01@hs-esslingen.de"
}
```

So basically the JWT Token will tell the server what the user is allowed to access.

Question: Can I not just modify my permission, e.g. my role to admin?
Answer: No, you can not, since the Token is signed with a private key from the Identity Provider (IdP). If you change the content, the signature will no longer match.

JSON Web Token vs. Opaque Token

Opaque tokens will be issued by the IdP and allow more control, but are harder to scale.
The Resource Server will then ask the IdP every time to validate the Opaque token. Similar to cookies.

Feature	JWT	Opaque Token
Structure	Self-contained token with claims (JSON data)	Random string with no inherent information

Feature	JWT	Opaque Token
Validation	Decoded and verified locally using a public key or secret	Requires introspection (remote check with auth server)
Performance	Fast (no network call needed for validation)	Slower (needs remote introspection)
Scalability	Better (stateless, no session storage)	Requires scalable auth server to handle introspection
Revocation	Harder (usually not revocable unless using short lifetimes)	Easier (server can revoke or blacklist token)
Security Risk	Exposed data if not encrypted; misuse if not expired properly	Minimal data exposure; relies on secure server
Typical Use Case	Microservices; high performance APIs	Centralized access control; stricter session control

---

PKCE

Ergänzen:

OAuth 2.1 extension: Do not use client secrets in SPA's, since they are public and would be easy to obtain.
Instead use PKCE.

angular-oauth2-oidc supports PKCE by default.

Keycloak supports PKCE.

plain vs S256

Configure Spring Boot with OAuth 2.0

Spring Boot OAuth 2.0 Resource Server

If you want to use API's on behalf of other users, for example GitHub or Ebay API, you would need Spring Boot OAuth Client.

In our case, the Spring Boot Server is the resource we want to protect, not some foreign API. This makes our Spring Boot App the "Resource Server". Therefore we need Spring Boot OAuth

Resource Server.

Additionally we are going to use JSON Web Tokens, not Opaque Tokens or custom JWT Tokens.

Add to pom.xml:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Add to application.properties:

```
# Keycloak settings
spring.security.oauth2.resourceserver.jwt.issuer-uri=https://auth.insy.hs-
esslingen.com/realms/insy
```

You can get the issuer-uri from the endpoints URI, listed at the top of this document.

or as yaml:

```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://my-auth-server.com
```

Add new package with SecurityConfig.java class:

```java
package com.hs_esslingen.insy.security;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.Customizer;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecu
```

```
rity;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.jwt.JwtDecoders;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
                .authorizeHttpRequests((authorize) -> authorize
                        .requestMatchers(HttpMethod.GET, "/login").permitAll()
                        .requestMatchers(HttpMethod.GET,
"/").hasAnyAuthority("ROLE_ADMIN", "ROLE_USER")
                        .anyRequest().authenticated()
                )
                .oauth2ResourceServer((oauth2) -> oauth2
                        .jwt(Customizer.withDefaults())
                );
        return http.build();
    }

    @Bean
    public JwtDecoder jwtDecoder() {
        return JwtDecoders.fromIssuerLocation("https://auth.insy.hs-
esslingen.com/realms/insy");
    }

}
```

- Evtl. **@EnableMethodSecurity** for more granular control. This will allow you to define restrictions directly on every route of a controller
    - with **@PreAuthorize**("hasRole('client_user')"), e.g.

SecurityFilterChain ???

Create controller package with new controller:

```
package com.hs_esslingen.insy.controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
@RequestMapping("/")
public class DemController {

    @GetMapping
    public String hello1() {
        return "Hello World!";
    }

    @GetMapping("/login")
    public String
    hello2() {
        return "Hello World from Keycloak!";
    }
}
```

Configure Angular with OAuth 2.0

Additional Documentation
AuthGuards)
Example Implementation

Available, well maintained packages: angular-oauth2-oidc or keycloak-js.

- OAuth dependency: angular-oauth2-oidc
- Angular Version: 19.2.6
- Node.js version: 22.13.1

OAuth Flow:

1. Install angular-oauth2-oidc
   `npm i angular-oauth2-oidc --save`
2. Add the following providers to main.ts (or app.config.ts):

   ```
   provideHttpClient(),
   provideOAuthClient()
   ```

3. Add the following to app.config.ts and adjust `YOUR_REAL_NAME` in `issuer`:

```
export const authCodeFlowConfig: AuthConfig = {
  // Url of the Identity Provider
  issuer: 'https://auth.insy.hs-esslingen.com/realms/YOUR_REALM_NAME',

  // URL of the SPA to redirect the user to after login
  redirectUri: window.location.origin + '/index.html',

  // The SPA's id. The SPA is registerd with this id at the auth-server
  // clientId: 'server.code',  clientId: 'spa',

  // Just needed if your auth server demands a secret. In general, this
  // is a sign that the auth server is not configured with SPAs in mind  //
and it might not enforce further best practices vital for security  // such
applications.  // dummyClientSecret: 'secret',
  responseType: 'code',

  // set the scope for the permissions the client should request
  // The first four are defined by OIDC.  // Important: Request offline_access
to get a refresh token  // The api scope is a usecase specific one  scope:
'openid profile email offline_access api',

  showDebugInformation: true,
};
```

4. When starting application, call:

```
constructor(private oauthService: OAuthService){
  this.oauthService.configure(authCodeFlowConfig);
  this.oauthService.loadDiscoveryDocumentAndTryLogin();

  // Token refresh upon 75% of tokens lifetime. Can be adjusted with
'timeoutFactor', from 0 to 1.
  this.oauthService.setupAutomaticSilentRefresh();
}
```

5. Call this, when logging in:

```
this.oauthService.initCodeFlow();
```

6. Call this, when logging out:

```
this.oauthService.logOut();
or
```

```
this.oauthService.revokeTokenAndLogout(); // revoke the existing access token
and the existing refresh token
```

Guards:
Guards only offer a visual layer of security. They restrict access to certain pages through the user interface, creating the impression that the user cannot access them. In reality, this mechanism simply blocks JavaScript from navigating to another HTML file — even though those files have already been fetched by the user's browser on the initial request to the resource server. This means all pages are technically accessible at any time. The only components that truly require protection are the APIs that serve the actual data.

1. Create new service and implement CanActivate Interface:

```
import { Injectable } from '@angular/core';
import {ActivatedRouteSnapshot, GuardResult, MaybeAsync, RouterStateSnapshot}
from '@angular/router';
import {AuthenticationService} from './authentication.service';

interface CanActivate {
}

@Injectable({
  providedIn: 'root'
})
export class DefaultGuardService implements CanActivate{

  constructor(private authService: AuthenticationService) {
  }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
MaybeAsync<GuardResult>{
    return this.authService.validToken();
  }
}
```

2. Add Guard to routes:

```
import { Routes } from '@angular/router';
import {InventoryComponent} from './pages/inventory/inventory.component';
import {HomepageComponent} from './pages/homepage/homepage.component';
import {NotFoundComponent} from './pages/not-found/not-found.component';
import {LoginComponent} from './pages/login/login.component';
import {DefaultGuardService} from './services/default-guard.service';
```

```
export const routes: Routes = [
  {
    title: 'Login',
    path: 'login',
    component: LoginComponent,
  },
  {
    title: 'Home',
    path: '',
    component: HomepageComponent,
    canActivate: [DefaultGuardService],
  },
  {
    title: 'Inventarliste',
    path: 'inventory',
    component: InventoryComponent,
    canActivate: [DefaultGuardService],
  },
  {
    title: '404 Not Found!',
    path: '**',
    component: NotFoundComponent,
  }
];
```

3. Voilá!

Configure Keycloak

A client is only needed for the Single Page Application (Angular App).

The Resource Server (Spring Boot) does not need to be registered as client, since it will just use Keycloaks public key and verify its issued JWT tokens.

Create Client:

1. Create new client
    1. General Settings
    2. Capability Config
        - Client Authentication OFF! (It's a public SPA and the client key will be publicly accessible anyway. That's why we are going to choose a much safer way: PKCE.)
        - Activate Standard Flow

        - Keep everything toggled off
    3. Login Settings
        - Set proper URI's
    4. Enforce PKCE: Go to Advanced -> Advanced Settings
        - Proof Key for Code Exchange Code Challenge Method to `S256`
    5. Done!

Create new users and groups:

Create client specific roles and map to groups:

Password policies:

Email Server:

Keycloak Documentation

[Server Administration]()

Important, may consider:

- SSL for Realm? Maybe this can be ignored with a Reverse Proxy?
- Email Server for a Realm? (Nutzer könne Passwörter zurücksetzen, Benachrichtigungen an Administratoren)
- Realm Settings
    - Enabling Forgot Password
    - Enabling Remember me
    - Update Email
    - Verify Email
    - Disable account deletion (Delete-account role)
    - Session and token timeouts
- Default Groups
- Password Policies
- OTP (FreeOTP or Google Authenticator)
- User session limits
- Brute Force protection

Negligible:

- Custom Theme
- Internationalization & Locale
- LDAP/Active Directory
- Agree to Terms & Conditions when registering
- Google reCAPTCHA
- Offline access (offline tokens)
- Transient sessions
- Rotating Realm keys
- Login Flow
- X.509
- Recovery Codes
- External Identity Providers (e.g. Facebook, Github, Gitlab, Google, LinkedIn, ...)
- SAML
- Docker registry v2 Authentication
- Realm Access Control (Allows to for realm admins to see only their own realm)
- Managing organizations and integrating their identities

Realm

Realms are isolated environments.

Use the Master realm only to create and manage other realms. Do not use it to secure your applications, create a new realm instead!

Managing Users

Defines a well-defined schema for representing user attributes.
Mainly targeted for attributes that end-users can change, e.g. last name, phone number etc.
But also allows administrators to:

- Enforce user profile compliance
- Define specific permissions for viewing and editing user attributes
- Define validation rules
- Dynamically render forms, that user interacts with
- Request user to reset password
- Setting required actions for one user or for all realm users
- Impersonate a user, to investigate a bug
- Revoke active sessions and tokens in case of compromised system

User Profile Contexts:

- Registration
- Update Profile
- Reviewing Profile when authenticating through aborker or social provider
- Account Console
- Administrative (e.g. administration console and Admin REST API)

You might restrict certain attributes to be available only from a certain context, e.g. only let user see certain attributes.

Managed and unmanaged Attributes:

- **Managed**: These are attributes controlled by your user profile, to which you want to allow end-users and administrators to manage from any user profile context. For these attributes, you want complete control on how and when they are managed.

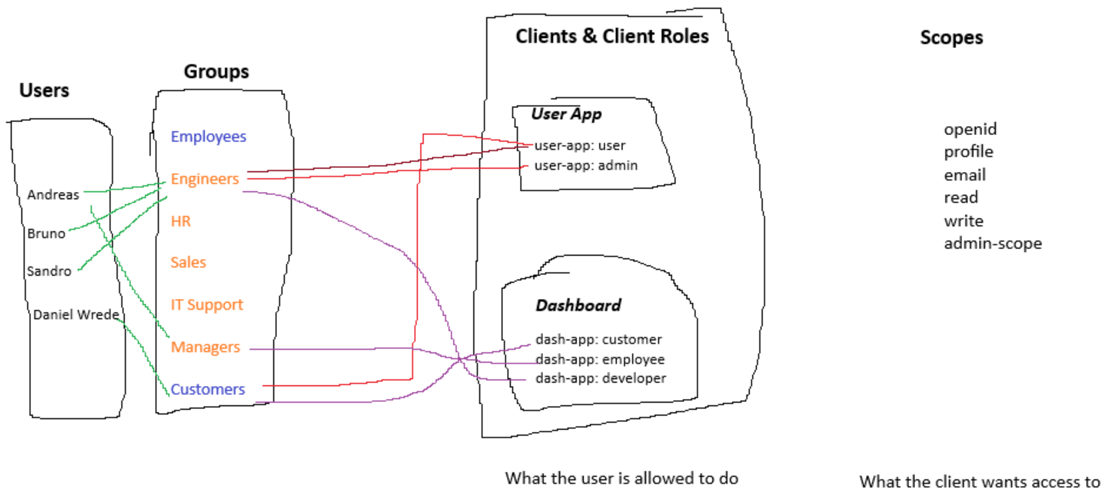
  Default attributes are username, email, firstName, lastName.
- **Unmanaged**: These are attributes you do not explicitly define in your user profile so that they are completely ignored by Keycloak, by default.
  - **Disabled**. This is the default policy so that unmanaged attributes are disabled from all user profile contexts.
  - **Enabled**. This policy enables unmanaged attributes to all user profile contexts.
  - **Admin can view**. This policy enables unmanaged attributes only from the administrative context as read-only.
  - **Admin can edit**. This policy enables unmanaged attributes only from the administrative context for reads and writes.

Attribute Groups allow to group attributes.

Validation & Permissions:
It is possible to set permissions and **Validators**, e.g. minLenght, maxLength, prohibited Characters.

Customizing attributes UI:
Annotations will allow to configure the HTML element of the formular, that will take the input, such as inputType, inputTypeMinLength, numberFormat, select-radiobuttons, multiselect-checkboxes, html5-range for a slider.

Assigning Permissions using Roles & Groups



Groups are a collection of users, to which you apply roles, e.g. `Employees` , `HR` , `IT Support` , `Managers` .
A role typically applies to one type of user, e.g. `admin` , `user` , `manager` and `employee` .

There is a global namespace for roles but also an **own dedicated namespace for each client,** where roles can be defined.

Realm Roles:
Namespace for defining your roles.

Client Roles:
Namespaces for roles dedicated to clients. Each client gets its own namespace.

Composite Roles:
Role that has one or more roles associated with it. Any realm or client level role can become a composite rule.
Recommended to not overuse this.

Default Roles:
It is possible to automatlly assign user role mappings to a user.

Role Mapping:
Assigning roles to a user or a group.

Role Scope Mapping:
Limit the roles declared inside an access token. When a client requests user authentication, the

access token it receives contains only the role mappings that are explicitly specified for the client's scope.

Application use access tokens to make decisions on resources controlled by the application. If an attacker obtains these tokens, he can use the permissions mapped to that token, to compromise your network.

By default, each client gets all the role mappings of the user.

Scopes:
Scopes will restrict the access of the client to a resource, by only giving it the necessary permissions, e.g. read scope.

This is likely unnecessary for our use case, as no third-party apps are accessing our resources. Additionally, our applications do not require granular access restrictions. For example, users either need to read or edit a small piece of data, which they will always need access to modify. Therefore, defining scopes for reading or writing doesn't make sense, as they are required all the time.

Scopes are only necessary for larger, more complex applications where a client may need access to only specific resources, depending on the user's use case.

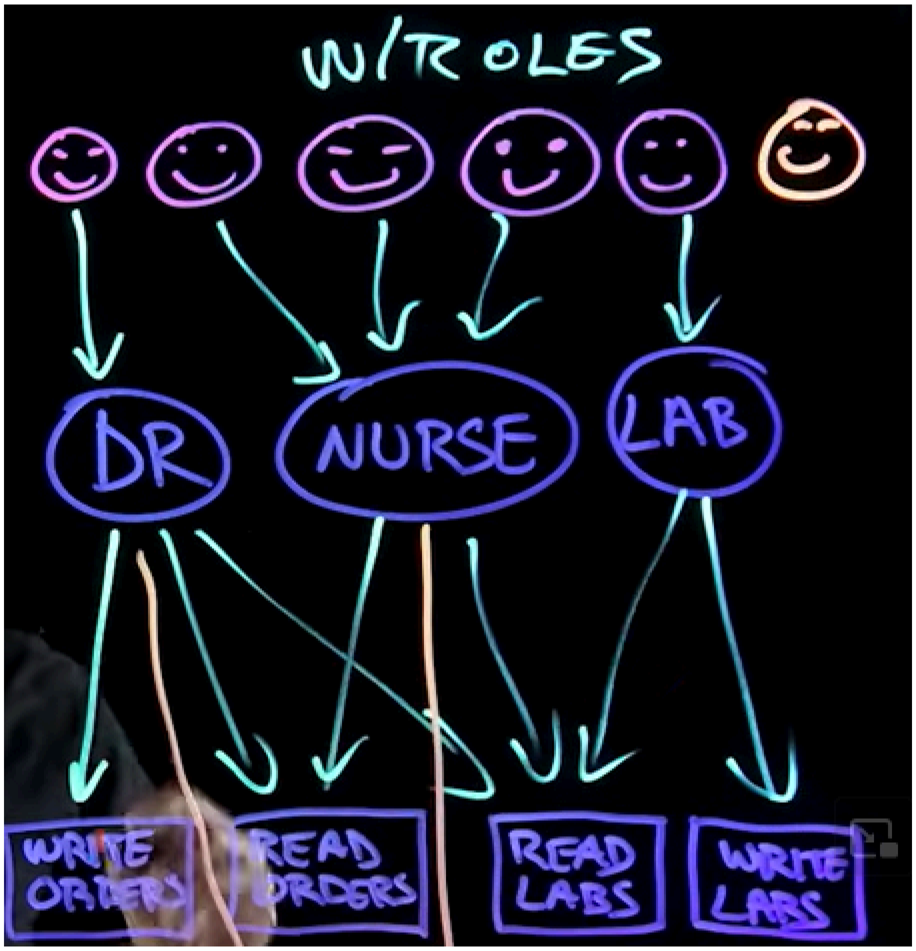
Roles should fit our use case perfectly fine.
By default full scope allowed.

Groups:
Users can be members of any number of groups. Groups are hierarchial and can be therefore inherited. A group can have multiple childs, but each group can only have one parent group.

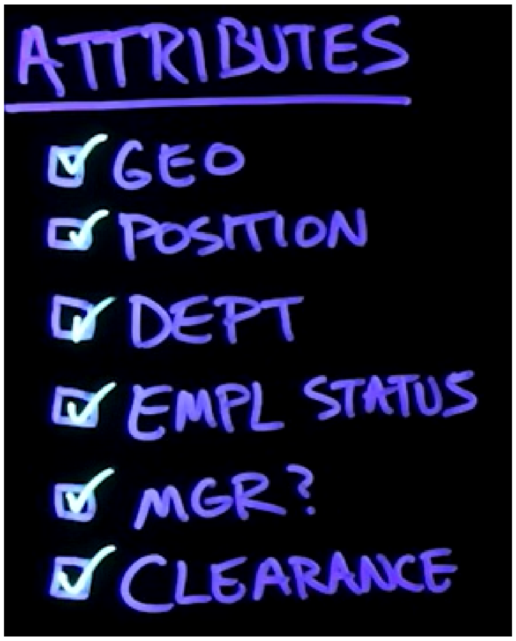
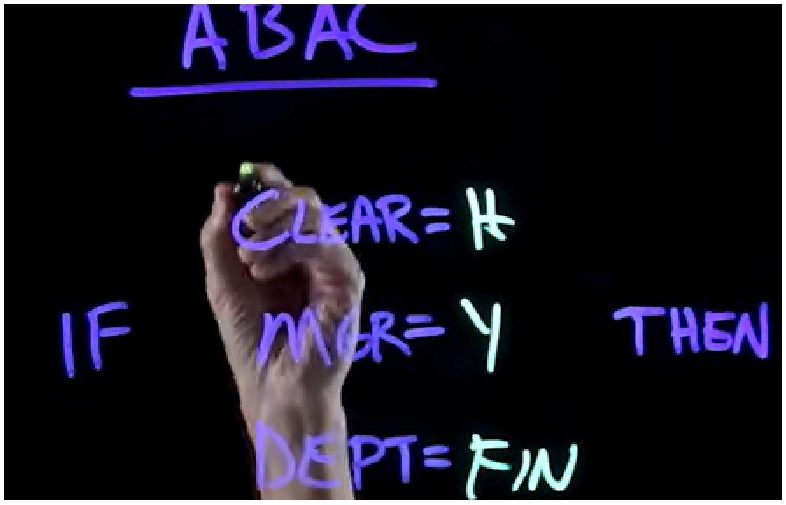
ENDE: Keycloak Server OIDC URI endpoints

RBAC vs. ABAC
Role-based access control (RBAC) will give users permissions depending on their role.

Better on hierarchical organisation.



Attribute-based access contol (ABAC) will give user permissions based on a set of attributes. Gives a lot more flexibility.

 ATTRIBUTES ☑ GEO ☑ POSITION ☑ DEPT ☑ EMPL STATUS ☑ MGR? ☑ CLEARANCE	 ABAC IF CLEAR=H MGR=Y DEPT=FIN THEN

Policy Enforcement Point (PEP)

Policy Decision Point (PDP)

Managing OpenID Connect and SAML Clients