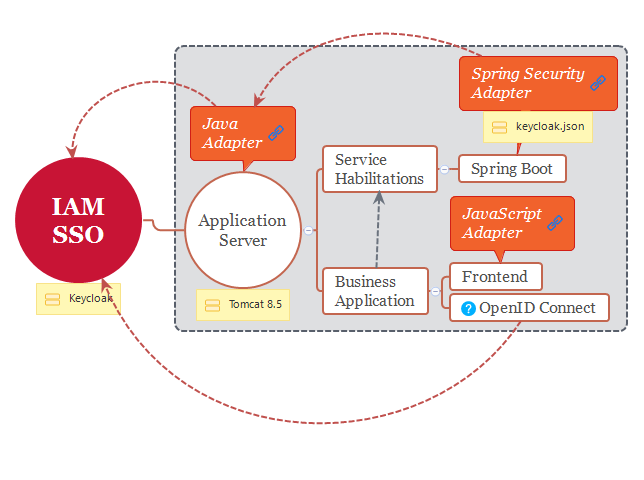
Introduction

Le service d'habilitations est un gestionnaire d'accès. Il doit s’appuyer sur un référentiel d'identification afin de partager avec les applications une base d'utilisateurs centrale.

Sur cette base, on peut donc imaginer trois scénarios de communication entre le service d'habilitations et ses clients (des applications dont on souhaite contrôler les accès) :

* Un annuaire de type [LDAP](https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol).
  + Dans ce cas, le client peut se connecter en mode "HTTP basic" au service qui va utiliser cet accès pour récupérer les informations complètes de l'utilisateur auprès de l'annuaire.
* Un serveur de type [Single Sign-On](https://en.wikipedia.org/wiki/Single_sign-on) (de type **Keycloak** ou Auth0).
  + Dans ce cas, toutes les informations d'identification sont transmises au service sous la forme d'un jeton encrypté (par exemple [JSON Web Token](https://en.wikipedia.org/wiki/JSON_Web_Token)).
* Une base interne d'utilisateurs, faisant ainsi du service de gestion d'accès un gestionnaire d'identités (ou *Identity and Access Manager*, [IAM](https://en.wikipedia.org/wiki/Identity_management)).
  + Ce scénario n'a pas été retenu, étant donné que des solutions standardisées de gestion d'identités existent sur le marché.

NB: le service s'appuyant sur une architecture **Spring Boot**, tous ces cas d'utilisation peuvent être pris en charge grâce à l'intégration des modules **Spring Security** correspondants.



Dans tous ces scénarios, on garde un concept commun : l'identification et les accès peuvent être gérés séparément... à condition de définir un pivot. Ce pivot, ce sont les rôles.

Dans les systèmes de gestion des identifications (dignes de ce nom), il est possible d'enregistrer des informations personnelles dans le compte des utilisateurs, en plus du *login*et du mot de passe, comme l'affectation de rôles d'utilisation (tels que USER, ADMIN, etc.). Ces rôles sont utilisés de manière classique dans les applications pour savoir ce que chaque utilisateur peut faire ou ne pas faire. Par exemple, pour accéder aux fonctionnalités ou aux détails de son compte, l'utilisateur doit être identifié et avoir le rôle USER. Mais pour changer la configuration de l'application ou accéder à certaines données des autres utilisateurs, un rôle différent, par exemple ADMIN ou MANAGER.

Cette approche, classique, on parle de Role-Based Access Control (RBAC), a deux lacunes importantes :

* Le périmètre des accès pour chaque rôle est statique, implanté dans le code source des applications.
* Du fait de son implantation dans le code source, le périmètre des accès reste opaque à l'administrateur.

C'est pourquoi il peut être intéressant de bénéficier d'un service externe, dans lequel les éléments applicatifs que l'on souhaite protéger sont déclarés, ce qui permet de le faire clairement. Dans cette solution, les accès accordés peuvent donc être non seulement rendus visibles à l'extérieur du code, mais également changer à n'importe quel moment, pour redéfinir au grès des besoin le périmètre des rôles accordés aux utilisateurs.

**Considérations importantes sur la sécurité applicative**

Le contrôle des habilitations est un élément qui peut mais ne doit pas se substituer à un contrôle des accès de bas niveau.

Même si l'on va parler de *ressources applicatives*, cette notion ne doit pas occulter les véritables ressources de l'application que son ses **fichiers statiques** (fichiers HTML, scripts, feuilles de style, etc.). Il est facile d'oublier ces "ressources" en ne se concentrant que sur les aspects ergonomiques de l'interface utilisateur, et pourtant le minimum que votre application doive sécuriser, avant même de parler de son comportement et de son système d'informations, ce sont :

* Les informations techniques de l'environnement d'exécution (système d'exploitation, machine virtuelle, serveur d'applications).
* Les fichiers source de l'application elle-même.

C'est pourquoi il faudra prendre soin de définir une protection de base (n'osons pas dire *minimale*) en exigeant :

* Que tout ce qui n'est pas nécessaire à la connexion ne soit accessible qu'une fois connecté au service ; y compris en faisant appel à de bons vieux filtres RBAC, pourquoi pas ?
  + Voir par exemple la section *security-constraint* dans le fichier *web.xml* de Java EE, ou le paramétrage via *HttpSecurity#authorizeRequests*de Spring Security.
* Que les éléments de journalisation du système (en particulier les informations techniques) soient à l'abri des regards d'un utilisateur quelconque.
* Que tous les paramètres par défaut du serveur aient été personnalisés lors de la mise en production (il serait trop bête que l'on puisse se connecter à sa console d'administration, n'est-ce pas ?)

Classification des éléments applicatifs

Imaginez une application, comme un site internet. Ce site est composé d'une page d'accueil, de modules fonctionnels divers (administration du compte utilisateur, saisie de documents, recherche de documents, organisation de réunions, saisie de notes de frais, etc.) et chaque module se compose de pages, onglets, formulaires de saisie, ou encore de boutons de déclenchement de traitements. Mais cette application se compose également d'un ou de plusieurs systèmes d'informations qui stockent et servent les données manipulées par l'interface.

Le service d'habilitations distingue donc deux types d'éléments :

* Les *Ressources*, qui représentent les éléments avec lesquels les utilisateurs inter-agissent directement dans l'application
  + Les modules (imaginez une section métier de l'application)
  + Les pages (par exemple l'écran d'un module regroupant une ou plusieurs activités)
  + Les écrans (qui sont une entité logique, un composant de saisie ; par exemple un formulaire ou un tableau)
  + Les commandes (qui sont une opération particulière, avec potentiellement un impact sur les données de plusieurs écrans ou périmètres métiers)
* Le *Data Realm*, qui représentent la structure des données à protéger ainsi que leurs traitements internes, comprennent...
  + Les schémas de base de données, qui regroupent les données par concept métier
  + Les tables et les colonnes dans lesquelles les données sont stockées
  + Les procédures stockées qui agissent sur les données directement à l'intérieur de leur base de données
  + A VENIR :
    - Les *data states*qui définissent une expression statique sur des colonnes afin de déterminer le statut des données (aka lignes, enregistrements)
      * Il est possible de définir pour une table une expression qui, appliquée à des données réelles, indique le *statut*des données
      * Lorsque les données sont bien dans le statut recherché, les droits accordés au niveau de la table peuvent être corrigés (GRANT, REVOKE)
    - Les *data ownerships*qui permettent de tracer dans quel contexte ont été enregistrées les données afin de définir des périmètres de contrôle
      * De la même façon, les données réelles peuvent être confrontées non plus aux droits de l'utilisateur mais à son appartenance à un *périmètre métier*
      * Lorsque les données correspondent au périmètre métier de l'utilisateur, les droits au niveau de la table peuvent être corrigés (GRANT, REVOKE)

Plusieurs remarques doivent être faites à propos de cette classification :

* Les types associés aux ressources ne sont pas figés (on peut en ajouter) et par ailleurs aucune règle de dépendance ne leur est appliquée.
  + Il n'y a pas de profondeur limite au graphe des ressources dans une application.
  + Les ressources peuvent être organisées indépendamment de leur type (un module sous une page, une commande sous module, etc.) ; c'est simplement une nomenclature facilitant leur organisation fonctionnelle.
* Par contre, la structure des graphes de data realm est beaucoup plus rigide.
  + Un table ou une procédure se trouvent forcément dans un schéma.
  + Une colonne se trouve forcément dans une table.
  + Les *data states* et les *data ownerships* sont également rattachés à une table (et plus précisément à ses colonnes).