Agence de la Biomédecine

DIRECTION DSI/Service SIAM

GUIDE DE DEVELOPPEMENT ANGULAR/SPRING REST

Version 2.4



agence de la biomédecine	**GUIDE DE DEVELOPPEMENT**	*Version 2.4*

Description du document	
Sujet du Document	**Guide de développement**
Date du document	**17/01/2022**
Nom du Document	**Guide de développement**
Auteur	**André-Pierre ABELLARD**
Version	**2.4**
Statut	

Diffusion du document					
Version	**Nom**	**Direction / Service**	**Pour validation**	**Pour information**	**Date**
2.3	Lionel Corlobé				11/09/2020

Validité du document			
Version	**Nom**	**Direction / Service**	**Date**

Historique du document		
Version	**Date**	**Description**
2.3	11/09/2020	

Sommaire

1 PRESENTATION GENERALE 8
1.1 OBJECTIFS DU DOCUMENT 8
2 ARCHITECTURE GENERALE 9
2.1 GESTIONNAIRE DE SERVICES 9
2.1.1 Concepts 9
2.1.2 Objectifs 9
2.1.3 WSO2 9
2.1.4 Fonctionnement général 10
2.2 AUTHENTIFICATION 10
2.2.1 Principe général 10
2.2.2 Mise en œuvre 11
2.3 AUTORISATIONS 11
2.3.1 Objectif 11
2.3.2 Frontend 11
2.3.3 Backend 11
2.3.4 Matrice des permissions 12
3 ARCHITECTURE DES SERVICES 13
3.1 ARCHITECTURES 13
3.2 SERVICES APPLICATIFS 13
3.2.1 Architecture type 13
3.2.2 Gestion des opérations CRUD 14
3.2.3 Passage des paramètres 14
3.3 SERVICES D'ACCES AUX DONNEES 15
3.3.1 Objectifs 15
3.3.2 Architecture 16
3.3.3 Entité ou DataModel 16
3.3.4 Interface JPA Repository 17
3.3.5 Data service 17
4 ERGONOMIE 18
4.1 OBJECTIF 18
4.2 ARCHITECTURE DE L'APPLICATION 18

4.2.1 Vue générale 18
4.2.2 Le bandeau de l'application 19
4.2.3 La barre d'outils 19
4.2.4 Le panneau latéral gauche 19
4.2.5 Le panneau principal 19
4.2.6 Le panneau latéral droit 20
4.3 GESTION DE L'AFFICHAGE DES ERREURS 21
4.4 TERMINOLOGIE 21
4.5 BOUTTONS : CHARTE DE COULEUR 21
5 DEVELOPPEMENT BACKEND 23
5.1 AVANT PROPOS 23
5.2 CONFIGURATION DU PROJET 23
5.2.1 Pile technique 23
5.2.2 Maven 23
5.2.3 Gestion des projets 29
5.2.4 Configuration de l'application 29
5.2.5 Variabilisation des développements pour la gestion des environnements 30
5.3 STRUCTURE DE L'APPLICATION 30
5.3.1 Les différents composants 30
5.3.2 Les packages 31
5.3.3 Les classes java 31
5.3.4 Les objets de la base de données 32
5.4 GESTION DE L'AUTHENTIFICATION 32
5.4.1 Transmission aux services du login de l'utilisateur. 32
5.4.2 Utilisation du login et du rôle 33
5.5 GESTION DES ACCES AUX RESSOURCES 34
5.5.1 Objectif 34
5.5.2 Gestion par Serveur d'autorisation 34
5.5.3 Gestion par l'application 34
5.6 RESTRICTION DE L'APPLICATION DES PERMISSIONS 34
5.6.1 Objectif 34
5.6.2 Implémentation 35
5.7 MISE EN PAGE ET DOCUMENTATION 36
5.7.1 Mise en page du code 36

5.7.2 Documentation et javadoc 36
5.8 GESTION DES ERREURS 36
5.8.1 Objectif 36
5.8.2 Les Exceptions 36
5.8.3 Service REST et Exception 38
5.9 GESTION DES LOG 39
5.9.1 Objectif de la log 39
5.9.2 Logger Java 40
5.9.3 Définition des niveaux de log 41
5.9.4 Message de log 43
5.10 DEFINITION D'UN CONTROLLER 44
5.10.1 Le controller 44
5.10.2 Les méthodes d'un controller 45
5.10.3 Identification de l'utilisateur 49
5.11 DEFINITION D'UN DATASERVICE 49
5.11.1 Le DataService 49
5.11.2 Méthode annotée @Transactional et accès base 49
5.12 DEFINITION D'UN REPOSITORY 50
5.12.1 Le Repository 50
5.12.2 Optimisation des recherches 50
5.12.3 Utilisation du findById 50
5.13 DEFINITION D'UNE ENTITE 51
5.13.1 Entité JPA 51
5.13.2 Définition des propriétés 52
5.13.3 DynamicUpdate 52
5.14 RECHERCHE D'ENREGISTREMENT 52
5.14.1 Objectif 52
5.14.2 Implémentation coté Service REST 52
5.14.3 Implémentation coté DataService 53
5.14.4 Implémentation coté Repository 53
5.14.5 Recherche du nombre d'enregistrements trouvés 53
6 DEVELOPPEMENT FRONTEND 54
6.1 AVANT PROPOS 54
6.1.1 Approche Générale : le framework Angular 54

6.2 CONFIGURATION DU PROJET 54
6.2.1 Pile technique 54
6.2.2 Installation du client Angular 55
6.2.3 Gestion des dépendances 55
6.2.4 Gestion des versions 56
STRUCTURE DE L'APPLICATION 56
6.3 56
6.3.1 Structure application web Angular 56
6.3.2 Structure du projet 56
6.3.3 Architecture des modules fonctionnels 57
6.3.4 Store 58
6.4 BONNES PRATIQUES 59
6.4.1 Convention de nommage 59
6.4.2 Typage 59
6.4.3 Pattern « Séparation des préoccupations » 60
6.4.4 Composant type Angular 60
6.4.5 Hook 60
6.4.6 Observables 61
6.4.7 IMMUTABILITE 61
6.4.8 GESTION DES CODES RETOUR HTTP 61
6.4.9 Accès au DOM – Data Binding 61
6.4.10 Gestion des évènements – Event Binding 62
6.4.11 Service 62
6.5 CREATION DES COMPOSANTS 62
6.5.1 Création de l'application 62
6.5.2 Création d'un module fonctionnel 63
6.5.3 Création d'un data model 63
6.5.4 Création d'une énumération 63
6.5.5 Création d'un service 63
6.5.6 Création d'une page/container 63
6.5.7 Création d'un fragment 63
6.6 GESTION DE L'AUTHENTIFICATION 64
6.6.1 Délégation de la gestion de l'authentification à la librairie Keycloak-js 64
6.6.2 Utilisation de la librairie Keycloak-js 64
}); 65

6.6.3 Injection du token d'authentification dans les requêtes http 65
6.7 GESTION DES ACCES AUX RESSOURCES 66
6.7.1 Objectif 66
6.7.2 Rôles et permissions 67
6.7.3 Masquage des éléments de l'interface 67
6.7.4 Routes Guard 68
6.8 LES ROUTES 69
6.8.1 Description 69
6.8.2 Définition des Router Outlet de l'application 70
6.8.3 Routing ForRoot 70
6.8.4 Routing ForChild 71
6.8.5 Utilisation des routes 73
6.8.6 Passage de paramètres 73
6.8.7 Lazy Loading 74
6.9 LES FORMULAIRES 76
6.10 GESTION DES ERREURS 77
6.10.1 Objectif 77
6.10.2 Implémentation 77
6.11 LES TESTS UNITAIRES 79
7 GESTION DE LA CONFIGURATION 80
7.1 FRONTEND 80
7.1.1 Gestionnaire des paquets 80
7.1.2 Gestion de la configuration de l'application 80
8 GESTION DE L'INITIALISATION DE L'APPLICATION 82

1 Présentation générale

1.1 OBJECTIFS DU DOCUMENT

L'objectif de ce document est de constituer un guide des bonnes pratiques de développement d'une application basée sur le socle technique Angular et Spring REST.

Il est constitué de plusieurs chapitres décrivant :

- La présentation de l'architecture générale nécessaire au fonctionnement d'une application Web basée sur des services.
- L'ergonomie décrivant la structure d'une application Angular, les recommandations en termes de charte de couleurs et de positionnement des différents éléments de l'interface, la standardisation des noms des actions.
- Les recommandations de développement de la partie Backend.
- Les recommandations de développement de la partie Frontend.
- La gestion de la configuration décrivant la gestion des dépendances Angular et l'externalisation de la configuration d'une application.



Enrichissement du document

Ce guide sera mis à jour au fil de l'eau pendant les développements.

2 Architecture générale

2.1 GESTIONNAIRE DE SERVICES

2.1.1 CONCEPTS

Les concepts liés à l'architecture orientée API sont les suivants :

- Séparation du Front (la partie visible de l'utilisateur) et du Back (le traitement serveur).
- Le Front et le Back sont liés par un contrat, la plupart du temps défini au sein d'une Gateway.
- La Gateway permet d'effectuer la médiation (routage, authentification, autorisations) entre le front et le back.
- Cette médiation permet :
    - Au front de rester totalement indépendant du back et d'avoir un point d'entrée unique.
    - Au back de n'avoir à se concentrer que sur son domaine métier.

2.1.2 OBJECTIFS

L'architecture orientée service permet une réalisation plus flexible des services associés à une application. Les objectifs de ce type d'architecture sont :

- Modularisation et granularité appropriée des services,
- Réutilisabilité et capacité à combiner plusieurs services au lieu d'en développer des nouveaux,
- Gestion facilitée des versions,
- Isolation des développements, des équipes spécifiques peuvent intervenir dans leur domaine de compétence.
- Spécification forcée des services si l'on veut les consommer par le front.

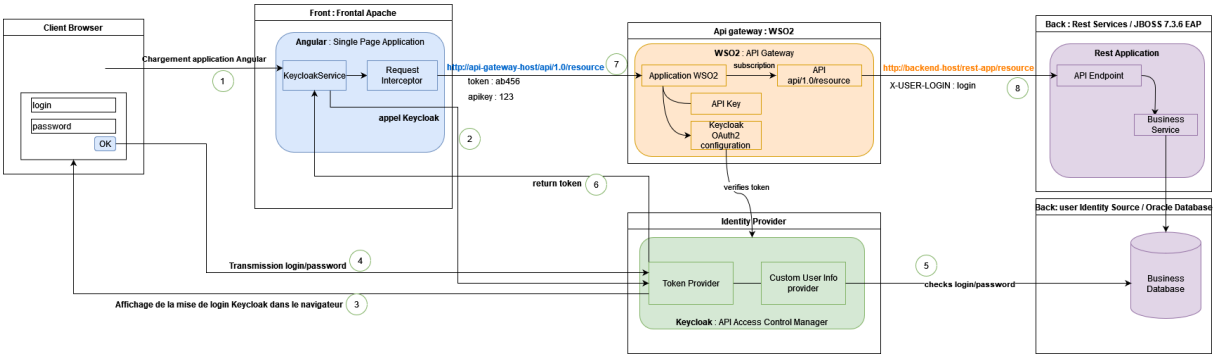
Les points sur lesquels il faut rester vigilant :

- Multiplication du nombre des services rendant la maintenance difficile,
- Multiplication du nombre des clients consommant ces services,
- Surveillance des performances, la modularité diminue l'adéquation entre le besoin et les services.

Cette architecture permet un urbanisme mieux maîtrisé, mais il faut poursuivre l'effort en permanence pour ne pas transformer le SI en « plat de spaghetti 2.0 ».

2.1.3 WSO2

L'architecture type mis en œuvre par la Gateway d'API (WSO2) pour l'Agence de la Biomédecine est la suivante :

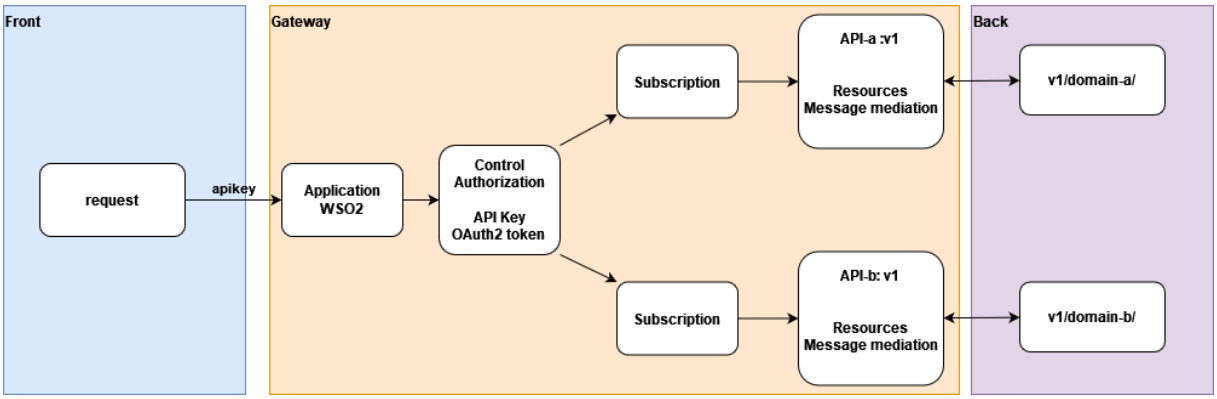


1. Chargement de l'application Angular dans le navigateur et initialisation de l'application.

2. Appel de l'Identity Provider Keycloak pour la transmission des éléments nécessaires à l'Authorization Code Flow.
3. Affichage de la mire de login Keycloak dans le navigateur.
4. Transmission des login/password à l'Identity Provider Keycloak.
5. L' Identity Provider possède un connecteur spécial pour la vérification de l'authentification et éventuellement des rôles associés à l'utilisateur.
6. Le token (jeton) est renvoyé au client, qui pourra l'envoyer à chaque requête. Le jeton a une durée de validité qui impose à l'utilisateur de se réauthentifier à l'expiration de ce jeton.
7. L'application Angular invoque une API au travers de la Gateway.
8. La Gateway vérifie le jeton puis route la requête ver le backend. Notez que c'est la Gateway qui transmet l'utilisateur authentifié au service, ce dernier s'occupe uniquement de gérer son métier.

2.1.4 FONCTIONNEMENT GENERAL

Schéma général



- **API** : permet d'identifier un service dans le backend, une fois les contrôles d'autorisation, des ressources et le message de médiation (optionnel) appliquées, la Gateway consommera le service pour le compte du client. Le client n'accède jamais au service sous-jacent en direct.
- **Control Authorization** : Permet de définir un ensemble de contrôle avant la consommation du service identifié par une API. Parmi ces contrôles, on peut retrouver les politiques :
  - **API Key** : Cross Origin Resource Sharing, symbolisant la stratégie d'ouverture des services invoqués à travers des domaines différents de celui de l'initiateur de la requête.
  - **Serveur d'autorisation (ici Keycloak)** : Gestionnaire d'authentification et d'autorisations
- **Subscription** (non présent sur le schéma) : La Subscription permet de définir des conditions d'utilisation des APIs fédérées par l'Application WSO2. Ces conditions d'utilisations peuvent notamment porter sur la limite d'invocation journalière d'une API (restriction d'usage).

2.2 AUTHENTIFICATION

2.2.1 PRINCIPE GENERAL

L'authentification a pour but de s'assurer que l'utilisateur à le droit d'accéder à l'application ; elle s'effectue en vérifiant que ce dernier, identifié par son login et son mot de passe, est bien connu par le système.

Méthodes d'authentifications

L'authentification peut être réalisée soit par un mécanisme interne propre à l'application, soit par un mécanisme externe de référentiels d'identités comme un LDAP ou Active Directory.

2.2.2 Mise en œuvre

Dans le cadre de la mise en œuvre de l'API Manager, la gestion de l'authentification est déléguée au module connecté au serveur d'autorisation. Ce dernier permet de s'appuyer sur les différents outils d'authentification du marché ou de mettre en œuvre des mécanismes spécifiques comme ce qui a été fait dans le cadre du Poc Green qui utilise le système de fonction stockée en base de données comme ce qui est fait pour les applications Cristal.

2.3 AUTORISATIONS

2.3.1 Objectif

Les autorisations ont pour vocation de dire ce que chaque utilisateur ou groupe d'utilisateurs ayant le même rôle, peut réaliser en termes d'action sur les objets fonctionnels de l'application, ces autorisations se déclinent en permissions basiques telles que :

- Peut créer,
- Peut modifier,
- Peut consulter,
- Peut supprimer.

Elles peuvent être complétées pour englober des permissions métiers comme par exemple :

- Peut approuver,
- Peut alerter,
- Etc.



Autorisation des actions

Toute action doit être soumise à permission.

Elles permettent aussi bien au Frontend qu'au Backend de s'assurer que l'utilisateur à le droit de réaliser l'action ciblée.



Attention

Les permissions sont entièrement découplées des données ; de ce fait, elles ne permettent pas de contrôler qu'un utilisateur peut par exemple modifier telle donnée selon le contexte.

2.3.2 Frontend

Pour le Frontend, la mise en œuvre des permissions a pour but de conditionner la navigation ou l'affichage / masquage de certains éléments graphiques comme les boutons, les liens ou des parties de l'interface de l'application.

2.3.3 Backend

Pour le Backend, cela se traduit par le contrôle des permissions avant la réalisation de l'action demandée et la transmission d'une erreur le cas échéant.

Ce contrôle peut être réalisé soit de manière spécifique au niveau du code d'exécution des contrôles de la couche service, soit de manière centralisée en s'appuyant sur le serveur d'autorisation.

Ce dernier, en plus de l'authentification, permet également de gérer les permissions d'accès aux API en définissant des rôles et en les appliquant aux chemins, verbes ou action mis en œuvre par les API.

2.3.4 MATRICE DES PERMISSIONS

Afin de déterminer les permissions à mettre en œuvre, il est nécessaire de connaitre les actions possibles par application, il est donc nécessaire d'élaborer une matrice des profils et des permissions sur le modèle suivant :

Fonctionnalité	Profil	
	SRA Médecin	SRA Secrétaire
Créer un incident	Oui	Oui
Modifier un incident	Oui	Oui
Supprimer un incident	Oui	Oui
Consulter un incident	Oui	Oui
Approuver un incident	Oui	Non
Alerter une équipe	Oui	Oui

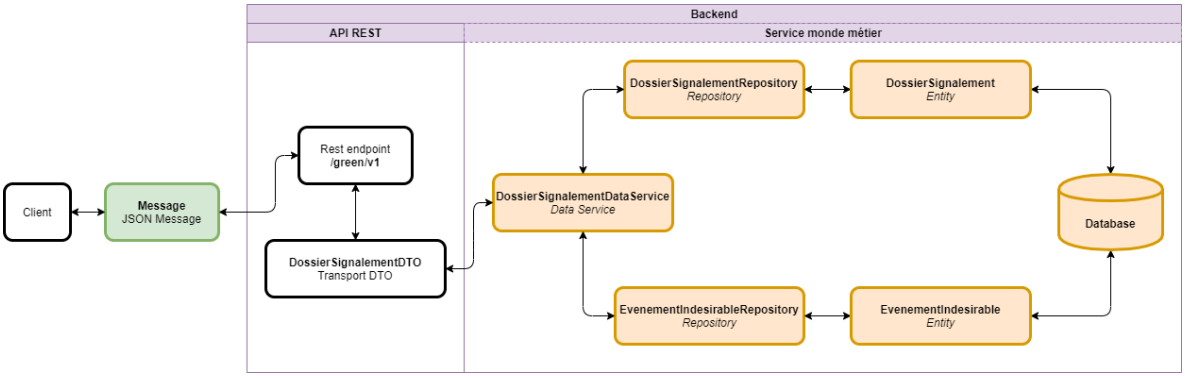
Restrictions complémentaires

Ces permissions peuvent s'accompagner de conditions supplémentaires pour un même rôle, comme par exemple une équipe de greffe qui ne peut modifier le dossier d'une autre équipe. Elles seront gérées en tant que règle métier par les API le nécessitant.

3 Architecture des services

3.1 ARCHITECTURES

Le schéma ci-dessous décrit l'architecture type mise en œuvre :



La couche de transport (API REST) fait le lien entre le service issu du domaine métier et le besoin de l'application. Ainsi, si une version différente du service applicatif est nécessaire, le monde métier n'est pas impacté. De même si le monde métier change, il est possible que le service applicatif parvienne à rendre cette modification transparente pour le client (pourvu que la modification ne soit pas incompatible avec le service jusqu'alors rendu).

3.2 SERVICES APPLICATIFS

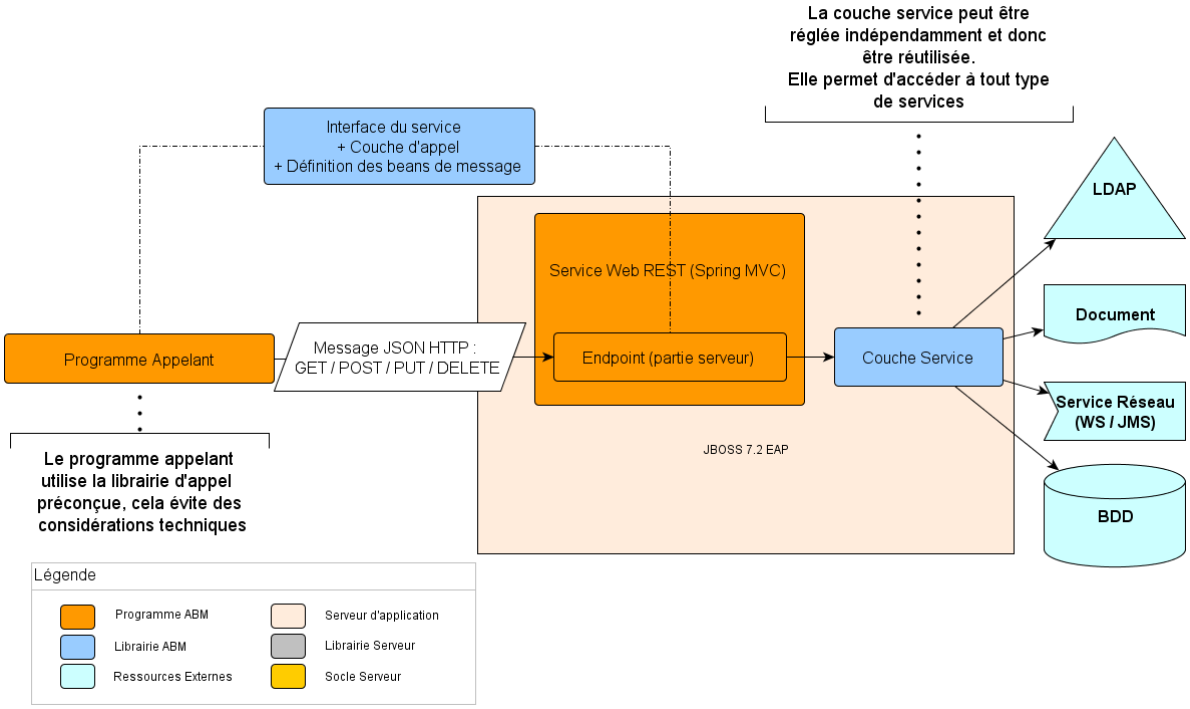
3.2.1 ARCHITECTURE TYPE

Les Services Web ***REST*** permettent d'exposer facilement des ressources au travers des méthodes prévues dans la norme ***HTTP***. L'acronyme REST signifie « **RE**presentational **S**tate **T**ransfer ».

Ce type de Service Web augmente la facilité d'accéder à des ressources, car contrairement à l'approche SOAP, il n'est pas nécessaire de définir un « contrat » (pour SOAP : WSDL) préalable à l'établissement de la communication. En réalité, les éléments sont requêtés par un simple appel d'url.

L'approche peut en revanche être vraiment identique, à ceci près que les échanges se font en fichier « plat » (traditionnellement ***JSON***), qui est une notation structurée et légère. C'est pourquoi il faut également une librairie pour manipuler ce type de données.

Le diagramme montre comment un appel REST peut se réaliser :



3.2.2 GESTION DES OPERATIONS CRUD

L'appel des services REST se fait en utilisant le protocole http ; il permet de normaliser les opérations CRUD sur les ressources en différentiant les opérations selon les méthodes http utilisées :

- GET : Lecture,
- POST : Création,
- PUT : Modification,
- DELETE : Suppression,
- PATCH : Mise à jour partielle.

3.2.3 PASSAGE DES PARAMETRES

3.2.3.1 Modalités http de passage de paramètres

Le protocole http permet le passage de paramètre de plusieurs manières possibles :

- Dans le corps de la requête : généralement sous forme de JSON
- Dans le chemin de la requête : /monservice/maressource/{id}
- Dans l'entête de la requête : généralement pour gérer les autorisations.
- Sous la forme de paramètre de requête : /monservice/maressource?nom=XXX&prenom=YYY

3.2.3.2 Passage d'un objet complexe

Tous les échanges devant véhiculer des objets complexes se feront via le corps de la requête et sous la forme d'un POST ou PUT. L'objet sera représenté au format JSON.



Passer une structure dans les cas des requêtes POST et PUT

POST /maressource HTTP/1.1

Content-Type: application/json

```
Content-Length: 48
Host: www.example.com
Accept :application/json; charset=UTF-8
{"name":"Jean","email":"jean.dupont@acme.com"}
```

3.2.3.3 *Passage de paramètres pour effectuer une recherche*

Tous les échanges de paramètre utilisés afin de rechercher des ressources se feront en passant les critères sous la forme de paramètre de requête et sous la forme d'un GET.



Passage de paramètres en tant que paramètres de requêtes pour les requêtes GET

```
GET /parents/children?age=15 HTTP/1.1
Host: www.example.com
Accept: application/json; charset=UTF-8
```

3.2.3.4 *Passage de paramètre pour récupérer ou supprimer une ressource*

Tous les échanges visant à récupérer ou supprimer une ressource selon son identifiant se fait en utilisant les paramètres de chemin et sous la forme d'un GET ou DELETE.



Passage de paramètres dans le chemin de la requête pour du GET ou DELETE

```
DELETE /parents/children/155 HTTP/1.1
Host: www.example.com
Accept: application/json;
```

3.3 SERVICES D'ACCES AUX DONNEES

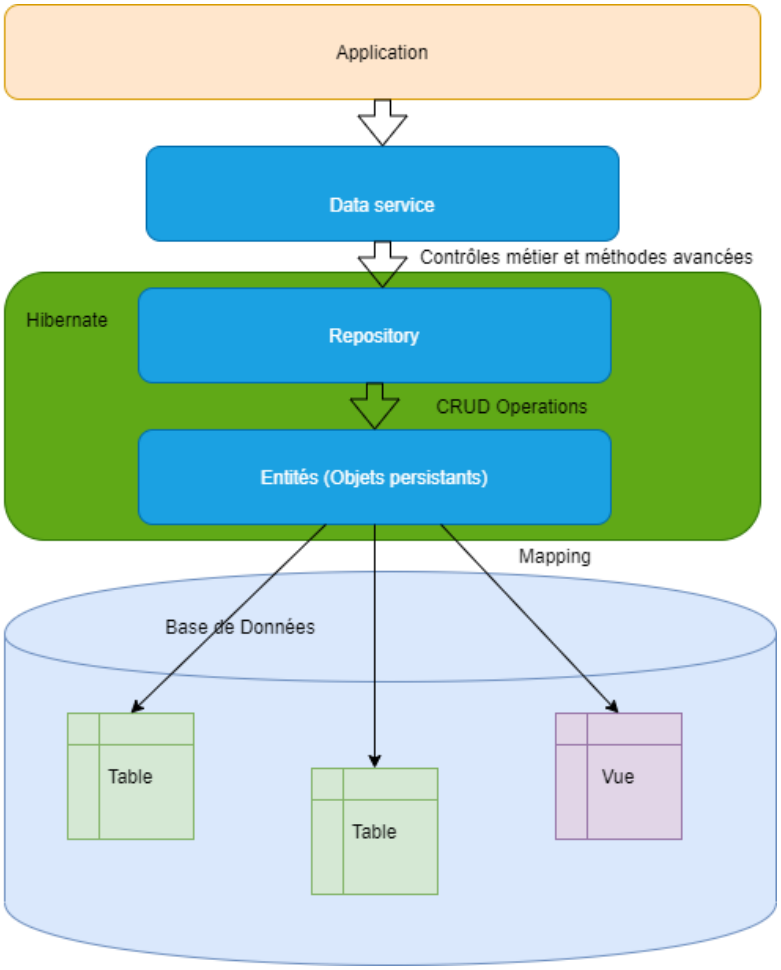
3.3.1 OBJECTIFS

La couche d'accès aux données a pour but de gérer la persistance des objets métier en base de données, et de permettre sa réutilisation par d'autres applications. Elle a aussi pour vocation à isoler la base de données en laissant les applications maitresses des données qu'elles veulent exposer.

La persistance est effectuée en se basant sur un framework de type ORM, **Hibernate**.

3.3.2 ARCHITECTURE

3.3.2.1 Schéma général



- Les entités : elles sont les objets métiers persistants.
- Les interfaces JPARepository : elles contiennent la définition de méthodes de bas niveau d'accès et de persistance des objets (sélection, insertion, mise à jour, suppression). Au besoin, elles permettent la définition de requêtes spécifiques de récupération d'une liste d'entités.
- Les data service : ils contiennent des méthodes plus avancées pouvant contenir des règles métiers.

3.3.3 ENTITE OU DATAMODEL

- Il mappe une table ou une vue de la base de données. Le mapping est réalisé en utilisant JPA (Java Persistent API).
- Il nécessite impérativement un identifiant simple (une colonne) ou composé (plusieurs colonnes) permettant l'identification d'un enregistrement de manière unique qu'il s'agisse d'une TABLE ou d'une VUE.
- Il permet d'expliciter le modèle de données en utilisant des noms de classe et d'attribut plus parlant.
- Il permet de réaliser des projections à partir du modèle de données.
- Il permet de définir de nouveaux sens de parcours, de nouvelles relations, entre entités.

3.3.4 INTERFACE JPA REPOSITORY

- JPA Respository est un module Spring permettant la réduction du code nécessaire aux accès bases de données. Il supprime le besoin d'un DAO.
- Une interface JPA Repository est définie pour une entité JPA, elle prend alors en charge sans écriture de code, les principales opérations de manipulations des tables de la base de données à savoir :
    - L'insertion d'enregistrement,
    - La mise à jour d'enregistrement,
    - La suppression d'enregistrement,
    - La recherche d'enregistrement.
- Si aucune transaction n'a été ouverte, les méthodes de l'interface, ouvrent une transaction vers la base de données afin de gérer les COMMIT ou ROLLBACK en cas d'erreur.

3.3.5 DATA SERVICE

- Il peut contenir des règles métier.
- Il peut agréger plusieurs JPA Repositories afin de réaliser des opérations complexes.
- Il est directement manipulé par les applications.
- Ses méthodes peuvent être annotée avec @Transactional pour la prise en charge des transactions.

4 Ergonomie

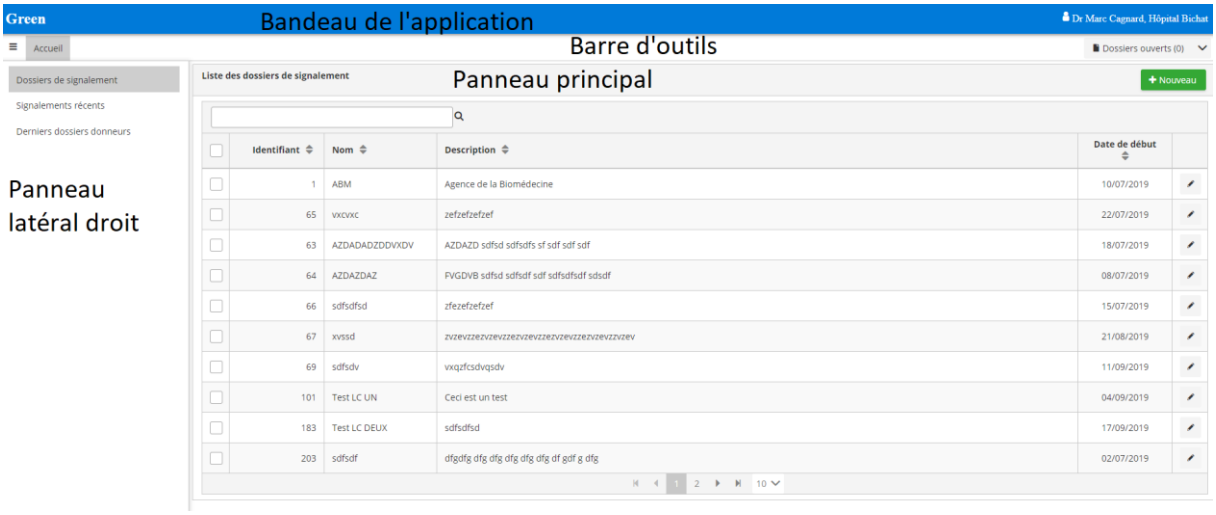
4.1 OBJECTIF

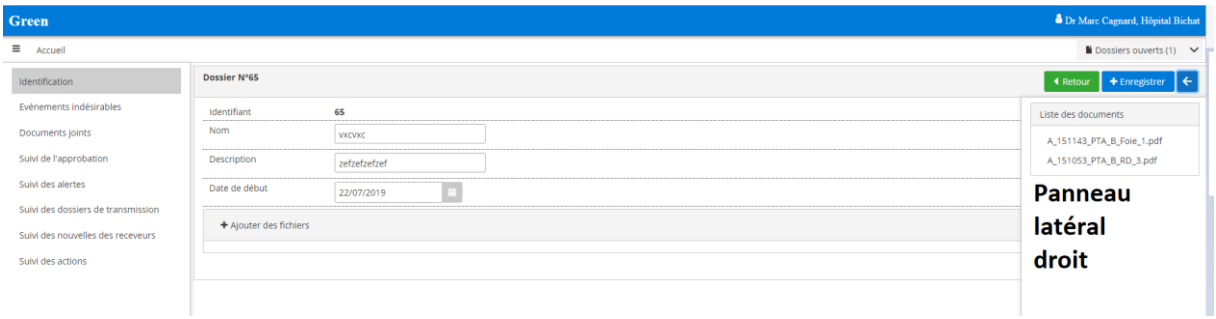
L'objectif de ce chapitre est de constituer un référentiel des règles d'ergonomie des applications développées en Angular / PrimeNg. Il décrit :

- L'architecture de l'application ;
- Charte de couleur ;
- Le nommage des principales fonctionnalités ;
- Les principaux composants.

4.2 ARCHITECTURE DE L'APPLICATION

4.2.1 VUE GENERALE





La structure de l'application se décompose en :

- Un bandeau principal ;
- Une barre d'outils ;
- Un panneau latéral droit ;
- Un panneau latéral gauche ;
- Un panneau principal.

4.2.2 LE BANDEAU DE L'APPLICATION

Le bandeau de l'application contient le nom de cette dernière, le logo de l'Agence et les informations de l'utilisateur connecté sous la forme de son titre, son prénom, son nom et son entité de connexion.

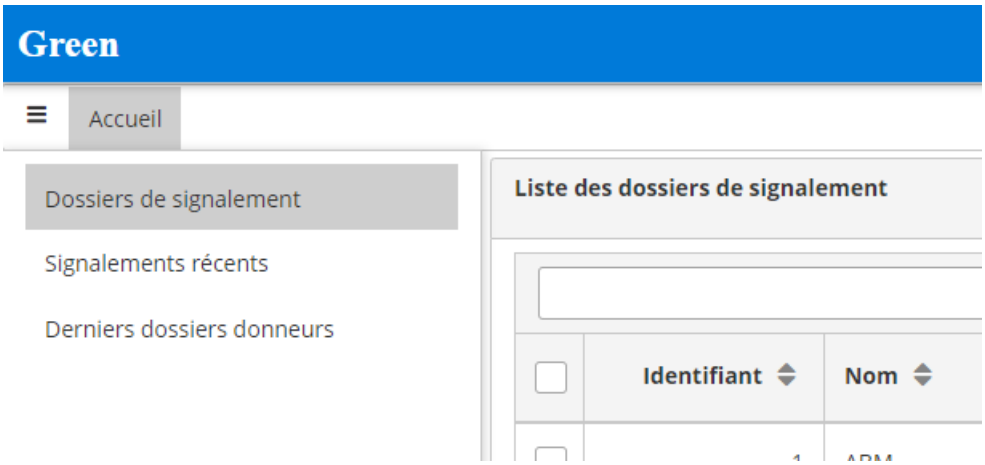
4.2.3 LA BARRE D'OUTILS

La barre d'outils contient :

Elément graphique	Description
≡	Le « hamburger » permettant d'afficher ou de masquer le panneau latéral gauche.
Accueil	Les principales fonctionnalités conditionnant le contenu du panneau latéral gauche.
Dossiers ouverts (1) N° 65	Un bouton permettant d'accéder rapidement aux dossiers ouverts.

4.2.4 LE PANNEAU LATERAL GAUCHE

Il contient les sous fonctionnalités liées à la fonctionnalité principale sélectionnée. En sélectionnant un élément de sa liste, il affiche le contenu dans le panneau principal :



Version de l'application

Le panneau latéral droit contient également la référence de l'application déployée ainsi que de l'environnement de la base de données :

Version 4.15.00 27/08/2019 ABMPDB

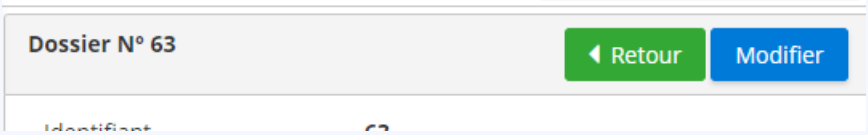
4.2.5 LE PANNEAU PRINCIPAL

Le panneau principal affiche les fonctionnalités de l'application sous la forme de :

- Listes ;
- Ecrans de consultation ;
- Formulaires de saisie.
- Graphiques.
- Etc.

Titre du panneau

Le titre du panneau doit refléter son contenu. Il contient également les boutons d'actions permettant d'agir sur son contenu général alignés à droite :



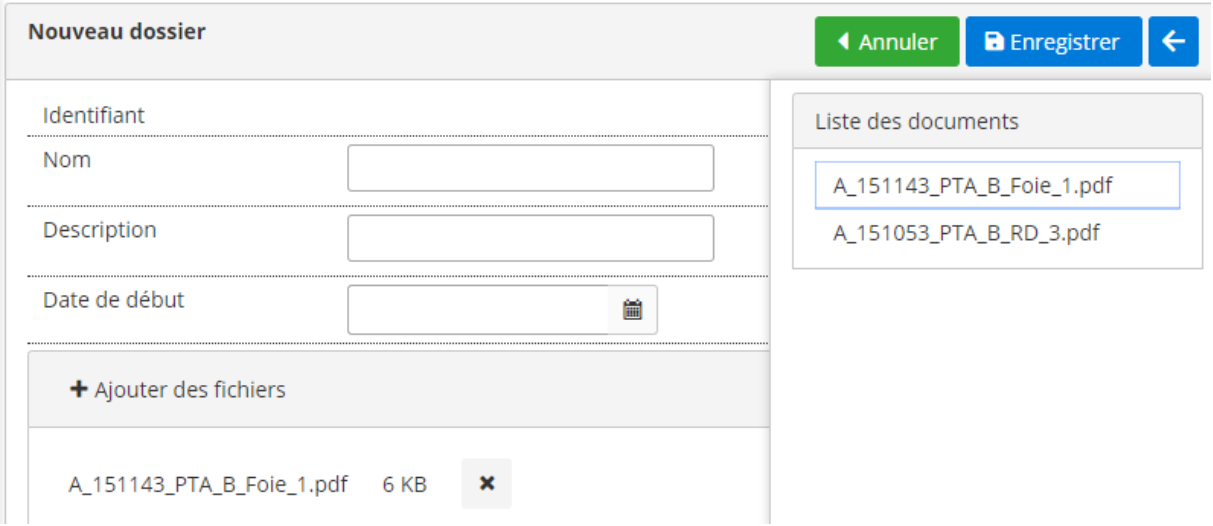
Contenu du panneau

Le panneau occupe la totalité de l'espace central et est fixe afin de permettre à l'utilisateur d'accéder en toute circonstance aux boutons d'action : si le contenu ne tient pas à l'écran, une barre de navigation latérale droite permet d'en faire défiler le contenu.

4.2.6 LE PANNEAU LATERAL DROIT

Le panneau latéral droit est contextuel à l'élément affiché dans le panneau principal.

Son ouverture s'effectue en cliquant sur le bouton .



Panneau latéral droit

Il permet d'afficher des informations complémentaires ou nécessaires au fonctionnement de l'élément affiché dans le panneau principal. Sa présence n'est pas systématique.

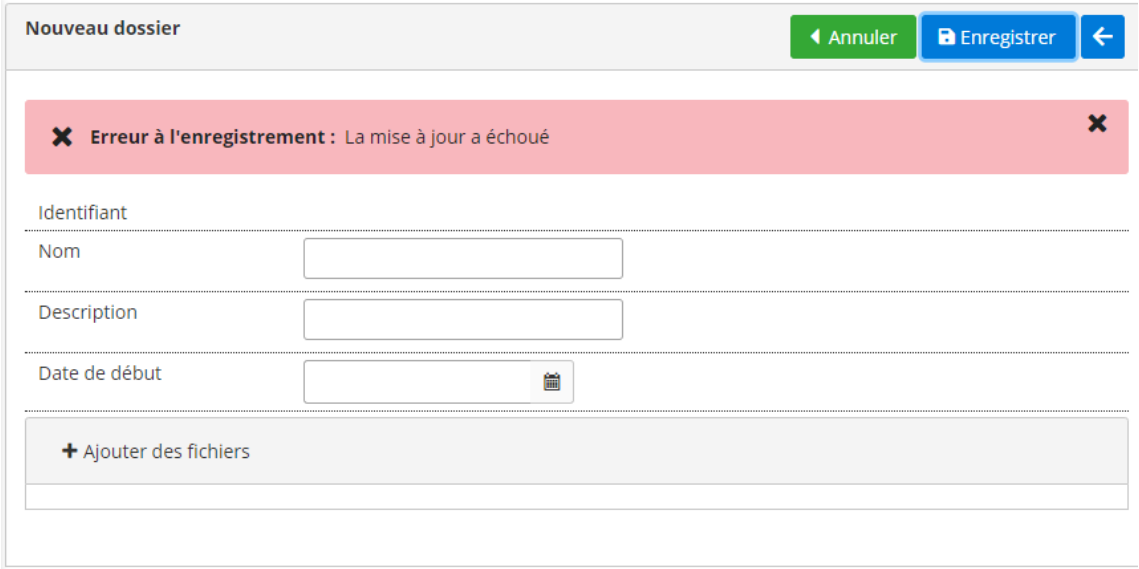
4.3 GESTION DE L'AFFICHAGE DES ERREURS

Deux types d'erreur doivent être affichées :

- Celles liées à des contrôles de formes appliqués à un champ de saisie ; elles sont affichées à la suite des champs :



- Ceux liés aux contrôles métier effectués lors de l'enregistrement d'un formulaire par exemple ; elles sont affichées dans le panneau central de l'application :



4.4 TERMINOLOGIE

Le style d'écriture est homogène et les termes employés sont les mêmes dans toutes les applications :

- « Enregistrer », plutôt que « Sauvegarder »,
- « Créer », plutôt que « Nouvel/Nouveau »,
- « Fermer », quand il s'agit d'une fenêtre ou d'un onglet,
- « Déconnexion », quand il s'agit de quitter l'application, plutôt que « Quitter »,
- « Annuler », quand il s'agit de quitter une création ou une modification, plutôt que « Abandonner »,
- « Précédent » quand il s'agit d'enchaîner les pages en sens inverse (ce qui nécessite « Suivant » à côté), plutôt que « Retour ».

On souligne qu'il existe de subtiles différences entre certains termes comme :

- « Accepter » : permet de valider à l'écran, sans enregistrement en base de données ;
- « Valider » : qui est réservé à la validation médicale ;
- « Enregistrer » : permet d'enregistrer en base de données.

4.5 BOUTTONS : CHARTE DE COULEUR

Il faut distinguer deux types d'actions dans les écrans :

- Les actions primaires entrainant un changement d'écran comme « créer », « enregistrer », « retour » ; les boutons doivent être visibles et à cet effet, le style « primaire » privilégié :



- Les actions secondaires du type sélection d'éléments liés à l'écran comme le bouton permettant d'afficher un calendrier, de sélectionner de la page dans une liste, etc.



5 Développement Backend

5.1 AVANT PROPOS



Langage utilisé

Le français sera utilisé pour nommer les objets, leurs attributs ou leurs méthodes. Le but est d'éviter toute interprétation.

5.2 CONFIGURATION DU PROJET

5.2.1 PILE TECHNIQUE

Pour l'Agence de la biomédecine, la pile technique pour le backend est la suivante :

- Spring MVC (partie REST)
- Spring Core 5.2
- JPA + Hibernate 5.4
- JBOSS 7.2 EAP + Java 11 pour le serveur



Versions de la pile technique

La version de la pile peut être amenée à évoluée selon les travaux de détermination du socle technique en cours à l'Agence.

5.2.2 MAVEN

Il est recommandé d'utiliser un pom parent permettant de définir une seule fois les numéros de version.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.abm.syrenad</groupId>
    <artifactId>syrenad-parent</artifactId>
    <version>1.00.00</version>
    <packaging>pom</packaging>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <springframework.core.version>5.2.5.RELEASE</springframework.core.version>
        <spring.jpa.version>2.2.0.RELEASE</spring.jpa.version>
        <hibernate.core.version>5.4.6.Final</hibernate.core.version>
        <hibernate.validator.version>6.0.2.Final</hibernate.validator.version>
        <log4j-core.version>2.17.1</log4j-core.version>
```

```
  </properties>

  <build>
    <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <version>3.2.1</version>
        <executions>
          <execution>
            <id>attach-sources</id>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          <javadocExecutable>${java.home}/bin/javadoc</javadocExecutable>
        </configuration>
        <executions>
          <execution>
            <id>attach-javadoc</id>
            <goals>
              <goal>jar</goal>
            </goals>
            <configuration>
              <additionalJOption>-Xdoclint:none</additionalJOption>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.2.3</version>
        <configuration>
          <archive>
            <manifestEntries>
              <Dependencies>jdk.unsupported</Dependencies>
            </manifestEntries>
          </archive>
        </configuration>
      </plugin>

    </plugins>
    </pluginManagement>
  </build>
  <dependencyManagement>
  <dependencies>
    <!--  Log4j -->
```

```
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-core</artifactId>
      <version>${log4j-core.version}</version>
    </dependency>
     <!--  Tests unitaires -->
    <dependency>
      <artifactId>junit-jupiter-api</artifactId>
      <version>5.6.1</version>
      <groupId>org.junit.jupiter</groupId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <artifactId>spring-test</artifactId>
      <version>${springframework.core.version}</version>
      <groupId>org.springframework</groupId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.jayway.jsonpath</groupId>
      <artifactId>json-path</artifactId>
      <version>2.2.0</version>
      <scope>test</scope>
    </dependency>
     <!--  Acces base de donnees -->
    <dependency>
      <groupId>javax.validation</groupId>
      <artifactId>validation-api</artifactId>
      <version>2.0.1.Final</version>
    </dependency>
    <dependency>
      <groupId>javax.inject</groupId>
      <artifactId>javax.inject</artifactId>
      <version>1</version>
    </dependency>
     <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-orm</artifactId>
      <version>${springframework.core.version}</version>
    </dependency>
    <dependency>
      <artifactId>spring-webmvc</artifactId>
      <version>${springframework.core.version}</version>
      <groupId>org.springframework</groupId>
    </dependency>
    <dependency>
      <artifactId>spring-web</artifactId>
      <version>${springframework.core.version}</version>
      <groupId>org.springframework</groupId>
    </dependency>
    <dependency>
      <artifactId>spring-context</artifactId>
      <version>${springframework.core.version}</version>
      <groupId>org.springframework</groupId>
    </dependency>
```

```
      <dependency>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-envers</artifactId>
         <version>${hibernate.core.version}</version>
      </dependency>
      <dependency>
         <groupId>org.springframework.data</groupId>
         <artifactId>spring-data-jpa</artifactId>
         <version>${spring.jpa.version}</version>
      </dependency>
       <dependency>
         <groupId>com.oracle</groupId>
         <artifactId>ojdbc8</artifactId>
         <version>12.2.0.1.0</version>
          <scope>provided</scope>
      </dependency>
       <dependency>
         <groupId>com.fasterxml.jackson.core</groupId>
         <artifactId>jackson-databind</artifactId>
         <version>2.9.9.3</version>
      </dependency>
      <dependency>
         <groupId>org.hibernate.validator</groupId>
         <artifactId>hibernate-validator</artifactId>
         <version>${hibernate.validator.version}</version>
      </dependency>
      <dependency>
         <groupId>org.hibernate.validator</groupId>
         <artifactId>hibernate-validator-annotation-processor</artifactId>
         <version>${hibernate.validator.version}</version>
      </dependency>
      <!-- Necessaire pour la validation et evite le probleme d'injection Spring : Unable to initialize
'javax.el.ExpressionFactory' -->
       <dependency>
         <groupId>org.glassfish.web</groupId>
         <artifactId>javax.el</artifactId>
         <version>2.2.6</version>
      </dependency>
      <dependency>
         <groupId>javax.interceptor</groupId>
         <artifactId>javax.interceptor-api</artifactId>
         <version>1.2.2</version>
         <scope>provided</scope>
      </dependency>
      <!--      Mapstruct -->
      <dependency>
       <groupId>org.mapstruct</groupId>
       <artifactId>mapstruct</artifactId>
       <version>1.3.0.Final</version>
     </dependency>
   </dependencies>
  </dependencyManagement>
</project>
```

Par la couche data service, la configuration Maven de la pile technique est la suivante :

```
<dependencies>
<!--  Log4j -->

<dependency>
   <groupId>org.apache.logging.log4j</groupId>
   <artifactId>log4j-core</artifactId>
 </dependency>
<!-- Log4j -->
<!--  Tests unitaires -->
<dependency>
   <artifactId>junit-jupiter-api</artifactId>
   <groupId>org.junit.jupiter</groupId>
  <scope>test</scope>
</dependency>
<dependency>
   <artifactId>spring-test</artifactId>
   <groupId>org.springframework</groupId>
   <scope>test</scope>
</dependency>

<!--  Acces base de donnees -->
<dependency>
   <groupId>javax.validation</groupId>
   <artifactId>validation-api</artifactId>
 </dependency>
<dependency>
   <groupId>javax.inject</groupId>
   <artifactId>javax.inject</artifactId>
</dependency>
  <dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-orm</artifactId>
</dependency>
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-envers</artifactId>
   <version>${hibernate.core.version}</version>
</dependency>
<dependency>
   <groupId>com.oracle</groupId>
   <artifactId>ojdbc8</artifactId>
   <version>12.2.0.1.0</version>
</dependency>
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
 </dependency>
 <!-- Validation des contraintes 5.0.2 (KO il ne trouve pas annotation-processor ou 6.0.2 -->
<dependency>
   <groupId>org.hibernate.validator</groupId>
   <artifactId>hibernate-validator</artifactId>
 </dependency>
<dependency>
   <groupId>org.hibernate.validator</groupId>
   <artifactId>hibernate-validator-annotation-processor</artifactId>
```

```
</dependency>
<!-- Necessaire pour la validation et evite le probleme d'injection SPring : Unable to initialize
'javax.el.ExpressionFactory' -->
 <dependency>
   <groupId>org.glassfish.web</groupId>
   <artifactId>javax.el</artifactId>
</dependency>
<dependency>
   <groupId>javax.interceptor</groupId>
   <artifactId>javax.interceptor-api</artifactId>
   <scope>provided</scope>
</dependency>

</dependencies>
```

Pour la couche REST, la configuration Maven de la pile technique est la suivante :

```
  <dependencies>
      <dependency>
        <artifactId>spring-webmvc</artifactId>
        <groupId>org.springframework</groupId>
      </dependency>
      <dependency>
        <artifactId>spring-web</artifactId>
         <groupId>org.springframework</groupId>
      </dependency>
      <dependency>
        <artifactId>spring-context</artifactId>
         <groupId>org.springframework</groupId>
      </dependency>

      <!-- Log4j -->
      <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
      </dependency>

      <!-- Log4j -->

      <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
      </dependency>
       <!-- Tests unitaires -->
      <dependency>
        <artifactId>junit-jupiter-api</artifactId>
        <groupId>org.junit.jupiter</groupId>
        <scope>test</scope>
      </dependency>
      <dependency>
        <artifactId>spring-test</artifactId>
        <groupId>org.springframework</groupId>
        <scope>test</scope>
      </dependency>
      <dependency>
        <groupId>com.jayway.jsonpath</groupId>
```

```
        <artifactId>json-path</artifactId>
        <scope>test</scope>
      </dependency>
      <!-- Acces base de donnees -->
      <dependency>
        <groupId>javax.validation</groupId>
        <artifactId>validation-api</artifactId>
      </dependency>
      <dependency>
        <groupId>javax.inject</groupId>
        <artifactId>javax.inject</artifactId>
      </dependency>
      <dependency>
        <groupId>org.hibernate.validator</groupId>
        <artifactId>hibernate-validator</artifactId>
      </dependency>
      <dependency>
        <groupId>org.hibernate.validator</groupId>
        <artifactId>hibernate-validator-annotation-processor</artifactId>
      </dependency>
      <!-- Necessaire pour la validation et evite le probleme d'injection SPring : Unable to initialize
'javax.el.ExpressionFactory' -->
      <dependency>
        <groupId>org.glassfish.web</groupId>
        <artifactId>javax.el</artifactId>
      </dependency>
      <dependency>
        <groupId>javax.interceptor</groupId>
        <artifactId>javax.interceptor-api</artifactId>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct</artifactId>
      </dependency>
    </dependencies>
```

5.2.3 GESTION DES PROJETS

La gestion des versions est effectuée via SVN.

- Un projet est créé pour la couche service REST : son nom reflète le domaine fonctionnel qu'il couvre, comme **« syrenad-donneur-rest »** par exemple. Le build de ce projet génère un fichier WAR.
- Un projet est créé pour la couche d'accès aux données : son nom reflète également le domaine fonctionnel qu'il couvre, comme **« syrenad-donneur-data »** par exemple. Le build de ce projet génère un fichier JAR qui sera inclus à la couche service.

Cette séparation permet de réutiliser la couche d'accès aux données dans d'autres applications.

5.2.4 CONFIGURATION DE L'APPLICATION

La connexion à la base de données doit être déclarée comme une ressource JNDI au niveau du serveur (JBOSS, etc.). Si plusieurs applications sont déployées au sein d'un même serveur, cela permet la centralisation de la déclaration de la datasource. De plus, la gestion des pools de connexion des serveurs est plus performante que celle embarquée par Hibernate.

Pour les logs :

- Il faut prévoir une rotation par date et par taille (idéalement une taille inférieure à 20 mo) avec un maximum de 10 fichiers.
- Le répertoire où se trouve ce fichier est configuré dans le fichier de properties de l'application.



Le logger sera configuré en production pour afficher les niveaux INFO et supérieurs.

5.2.5 VARIABILISATION DES DEVELOPPEMENTS POUR LA GESTION DES ENVIRONNEMENTS

Il est important de ne pas mettre les valeurs des variables dans le code mais de les variabiliser dans des fichiers annexe afin de rendre le code plus lisible.

Pour le Back End, 2 stratégies sont possibles :

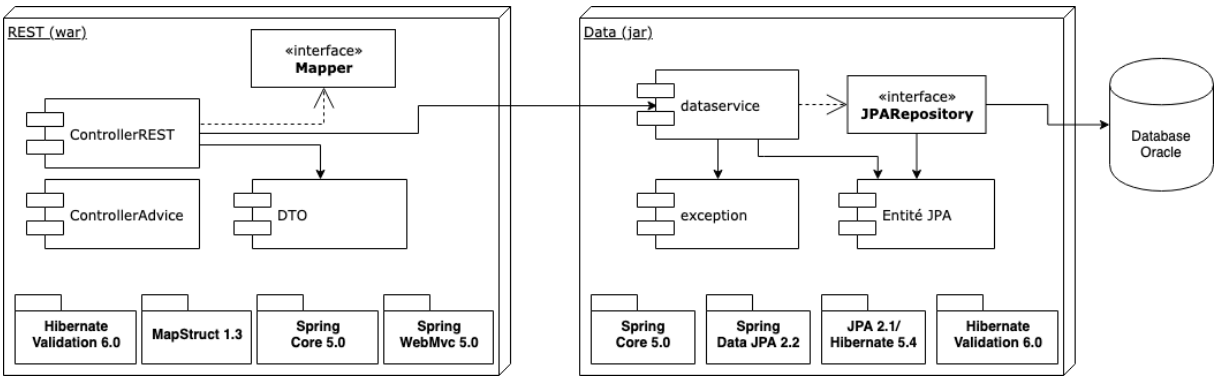
- Utiliser des binding JNDI pour découpler l'application de sa configuration. Cette approche est à privilégier, notamment pour toutes les services mise à disposition par JBoss (par exemple les Datasource, file JMS, etc.). Elle ne nécessite pas de modification des scripts de démarrage du serveur, les configurations de différentes applications peuvent facilement être mutualisées ou isolées en fonction des besoins.
- Sinon si la mise en place de binding JNDI ne peut pas être utilisé, alors utiliser des fichiers de properties, qui seront chargés au démarrage du serveur d'application (voir fichier de configuration XML),

5.3 STRUCTURE DE L'APPLICATION

5.3.1 LES DIFFERENTS COMPOSANTS

La partie back est décomposée en deux modules : la partie REST et la partie DataServices (voir chapitre 3). On aura ainsi deux projets Java :

- Un projet rest qui produit le war
- Un projet dataservice qui produit un jar qui sera utilisé par le projet rest et qui sera inclut dans le war



5.3.2 LES PACKAGES

Ils suivent le regroupement par fonctionnalité et par couche de composants.



Le nom des packages doivent être :

- **En minuscules**
- **Au singulier**

Pour la partie Data service :

```
com.abm.monappli.data.mondomainefonctionnel.repository
com.abm.monappli.data.mondomainefonctionnel.model
com.abm.monappli.data.mondomainefonctionnel.service

com.abm.monappli.data.core.exception

com.abm.monappli.data.core.service
```

Où :

- repository contient les interfaces qui héritent de JPARepository
- model contient les classes entités JPA et les classes Embdeddable
- dataservice contient les classes de type DataServices
- le package core.exception contient les exceptions du projet

Pour la partie REST :

```
com.abm.monappli.rest.mondomainefonctionnel.dto
com.abm.monappli.rest.mondomainefonctionnel.controller

com.abm.monappli.rest.mondomainefonctionnel.mapper
```

Où

- controller contient les controllers REST
- dto contient les DataTransfertObject qui correspondent aux messages json échangés avec le front
- mapper contient les interfaces Mapper utilisées par les controllers REST pour transformer les dto en entité JPA.

5.3.3 LES CLASSES JAVA

Les noms des classes et interfaces java doivent respecter les conventions suivantes :

- Le nom des classes, méthode et variables doivent être en français pour le domaine fonctionnel afin d'éviter des écarts de compréhension avec le métier
- Les noms des classes doivent utiliser la notation « Pascal case » (NomDeClasse).
- Les noms des classes qui sont des contrôleurs doivent être suffixés par « Controller ». Exemple : UtilisateurController.java
- Les noms des classes qui sont des DataTransferObject doivent être suffixés par « DTO ». Exemple : UtilisateurDTO.java

- Les noms des interfaces utilisées par MapStruct pour réaliser le mapping entre un DTO et une entité JPA doivent être suffixés par « Mapper ». Exemple : UtilisateurMapper.java
- Les noms des classes qui sont des data services doivent être suffixés par « DataService ». Exemple : UtilisateurDataService.java
- Les noms des interfaces qui héritent de JPARepository doivent être suffixés par « Repository ». Exemple : UtilisateurRepository.java
- Les noms des classes qui sont utilisées comme EmbeddedId par les entités JPA pour définir les clés composites doivent être suffixés par « PK ». Exemple : UtilisateurPK.java
- Les noms des classes qui implémentent des interfaces doivent être suffixés par « Impl ». Exemple : UtilisateurServiceImpl.java
- Les noms des classes qui sont des Enums doivent être suffixés par « Enum ». Exemple : CodeEnum.java
- Les noms des classes qui sont des exceptions doivent être suffixés par « Exception ». Exemple : UserNotFoundException.java
- Les noms des classes utilitaires doivent être suffixés par « Utils ». Exemple : DateUtils.java



Les noms des méthodes et propriétés doivent utiliser la notation « lowerCamelCase »

On commence par une minuscule, ensuite chaque nouveau mot constituant le nom commence par une majuscule : nomDeMethode.



Les noms des constantes doivent être en majuscule et les mots séparés par des underscores (SREAMING_SNAKE_CASE)

5.3.4 LES OBJETS DE LA BASE DE DONNEES

De nombreux projets réutilisent un modèle de données physique existant, dans ce cas, ils devront respecter la convention en usage pour ce modèle de données.

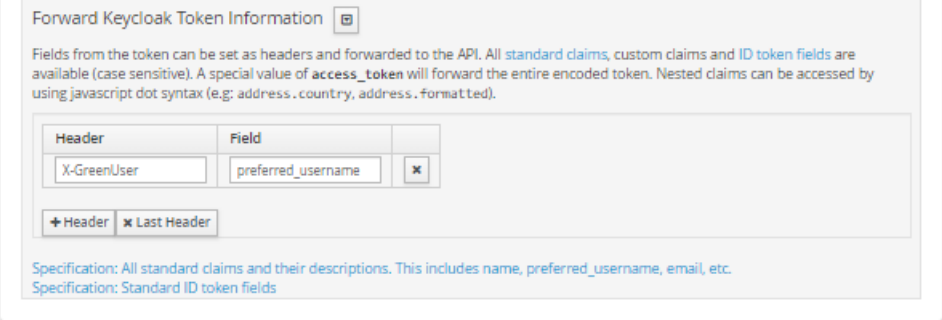
En cas de création d'un nouveau modèle de données, il faudra respecter la convention suivante :

- Les noms des tables, vues, colonnes, clés étrangères et indexes doivent être en français.
- Les noms des vues doivent être préfixés par « V_ ».
- Les noms des clés étrangères doivent être préfixés par « FK_ ».
- Les noms des indexes doivent être préfixés par « IDX_ ».

5.4 GESTION DE L'AUTHENTIFICATION

5.4.1 TRANSMISSION AUX SERVICES DU LOGIN DE L'UTILISATEUR.

L'authentification est déléguée au serveur d'autorisation via l'API Manager. Si elle réussit, le serveur d'autorisation transmet alors systématiquement aux services le login de l'utilisateur en tant que paramètre de Header de la requête http ; ceci est effectué en paramétrant L'API Manager (ici APIMan) de la façon suivante :



Ici, le login est transmis via le paramètre « X-GreenUser » aux services du Backend.

Ce mapping est défini dans l'API Manager et dépend de l'application. Il correspond à X-USER-LOGIN défini après. L'unification du nom de cet header est souhaitable mais en fonction des besoins spécifiques, il peut être amené à changer, en accord avec le responsable projet.

Transmission du rôle

La transmission du rôle est dédiée à la partie Frontend sous la forme d'injection dans le header des requêtes via l'attribut «X_USER_ROLE» , voir le chapitre dédié.

5.4.2 UTILISATION DU LOGIN ET DU ROLE

X-USER-LOGIN est le header générique permettant de récupérer de login de l'utilisateur au niveau de l'Agence (Portail SIPG).

Ces derniers sont récupérés en tant qu'header de la requête dans les services :

```
/**
 * Paginated and sorted search
 */
@GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
public List<ProjetMessage> getAllProjets(
@RequestHeader ("X-USER-LOGIN")String user,
        @RequestParam(name = "limit", required = false) int limit,
        @RequestParam(name = "offset", required = false) int offset,
        @RequestParam(name = "sort_order", required = false) String sort_order)  {
final Projet = toProjet(new ProjetMessage());
final List<Projet> projetList = projetDataService.getListProjets(projet, offset, limit, sort_order);
final List<ProjetMessage> projetMessageList = toProjetMessageList(projetList);
        return projetMessageList;
}
```

Remarque : d'autres paramètres peuvent être ajoutés depuis l'application afin de permettre des restrictions complémentaires.

5.5 GESTION DES ACCES AUX RESSOURCES

5.5.1 OBJECTIF

Pour une question de sécurité, l'accès aux API doit être conditionné au rôle de l'utilisateur connecté, voir, de manière plus fine, à des permissions que ce rôle peut lui octroyer.

A cet effet, la délégation des contrôles d'accès peut être faite soit auprès du serveur d'autorisation, soit gérée directement dans l'application.

5.5.2 GESTION PAR SERVEUR D'AUTORISATION

Le serveur d'autorisation, via son service de gestion des autorisations, offre plusieurs types de stratégies d'accès aux API :

- Par rôle,
- Par permission,
- Par moteur de règles,
- Par attribut,
- Par contexte,
- Par limitation dans le temps.

Il permet également de définir son propre mécanisme de contrôle des accès à travers une interface dédiée (SPI).



Avantage d'un serveur d'autorisations

Il permet de centraliser la gestion des rôles et des permissions et de décharger le développeur de devoir le faire manuellement pour chaque API exposée.

5.5.3 GESTION PAR L'APPLICATION

Dans le cadre d'une gestion des permissions par l'application, ceux sont les services d'accès aux données qui ont la charge de réaliser ces opérations.

5.6 RESTRICTION DE L'APPLICATION DES PERMISSIONS

5.6.1 OBJECTIF

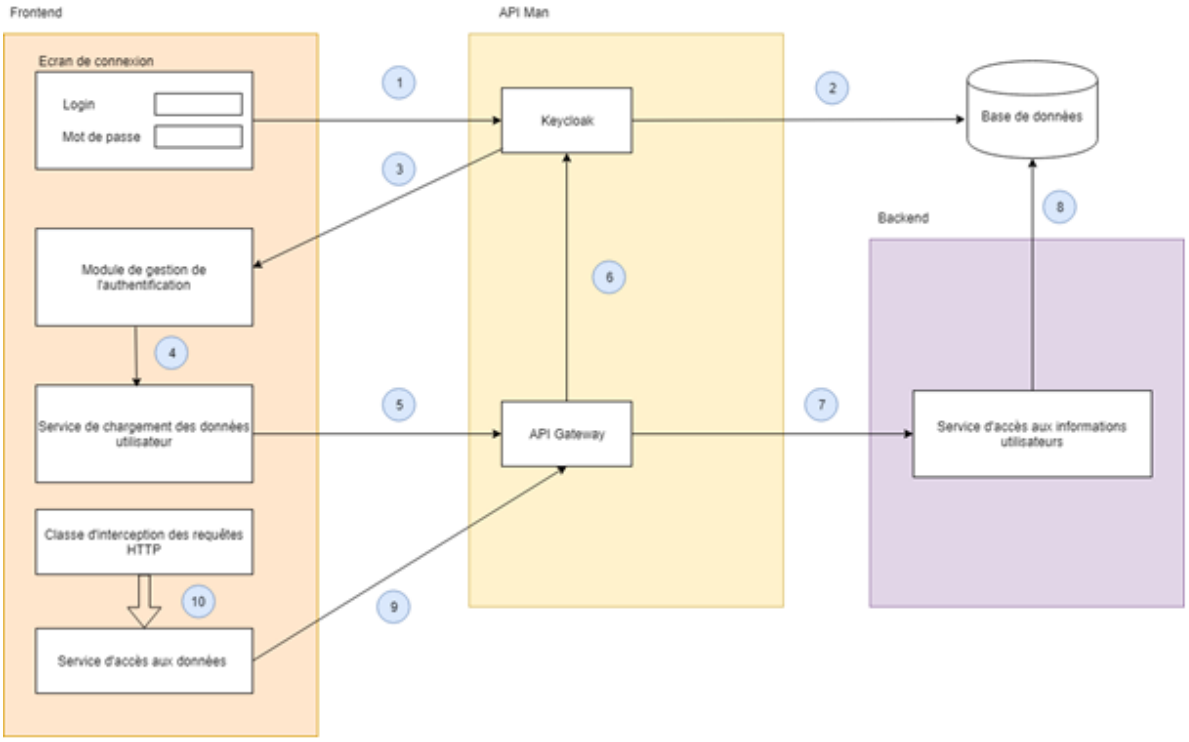
Il peut être nécessaire d'ajouter des contrôles métiers lors de l'appel des API afin de restreindre une permission ; par exemple, une équipe de greffe qui a la permission de modifier un dossier d'attente, ne pourra le faire que si ce dernier lui appartient.

Ce type de restriction sera géré par la couche métier de l'application sur la base des informations suivantes :

- L'utilisateur connecté : identifiant.
- Les données accédées.
- Le contexte utilisateur (entité de connexion, zone géographique, etc.)

Toutes ces informations doivent être transmises au service REST afin que ce dernier puisse les transmettre à la couche service d'accès aux données qui vérifieront que l'utilisateur à le droit d'effectuer l'action.

5.6.2 IMPLEMENTATION



1) A la validation du formulaire de saisie du login et mot de passe, l'application appelle le serveur d'autorisation chargé de vérifier l'authentification.
2) Cette vérification est effectuée en appelant la procédure stockée SESAME dédiée.
3) Si le contrôle est ok, le serveur d'autorisation retourne un token d'accès à l'application.
4) L'application appelle alors le service Angular de récupérations des données utilisateur.
5) Le service Angular appelle le service du backend via l'API Manager.
6) L'API Manager vérifie auprès du serveur d'autorisation la validité du token d'accès.
7) Si le token est valide, le service de chargement est appelé.
8) Le service charge les données utilisateurs de la base de données et les retourne à l'application.
9) L'application récupère des informations métier.
10) La classe d'interception injecte les informations liées au contexte utilisateur qui sont transmises au service du backend.

Protection du contexte utilisateur

Il est nécessaire d'encoder et de contrôler le contexte utilisateur afin d'empêcher sa compromission coté client : son entité de connexion ne doit pas pouvoir être modifiée à la volée coté client.
Exemple de contexte client : {'role' :'E','entite' :'PI7RA'}

5.7 MISE EN PAGE ET DOCUMENTATION

5.7.1 MISE EN PAGE DU CODE

Les développeurs doivent s'assurer que le code source est formaté conformément au fichier de formatage de l'IDE, avant de commiter.

Le fichier de formatage de l'IDE doit contenir les règles suivantes :

- 120 caractères par ligne
- Espaces pour indenter, plutôt qu'une tabulation
- Style d'indentation : K&R, variante 1TBS

5.7.2 DOCUMENTATION ET JAVADOC

Toutes classes et méthodes publiques doit disposer d'une documentation au format Javadoc.

5.8 GESTION DES ERREURS

5.8.1 OBJECTIF

Une application doit faire face à différents types d'événements qui interrompent son fonctionnement standard :

- Les actions utilisateurs incorrectes comme la saisie d'une valeur incorrecte ou la non saisie d'une valeur obligatoire,
- Les manques de ressource du système (OutOfMemoryError…) ou une configuration incorrecte
- Les erreurs de programmation comme un NullPointerException…

Le programme doit :

- Être capable d'informer correctement l'utilisateur sans dévoiler des informations techniques qui pourrait aiguiller les attaques d'un hacker,
- Permettre à l'exploitation et au développeur de corriger l'erreur s'il s'agit d'un problème de ressource, configuration ou de développement.

5.8.2 LES EXCEPTIONS

Le mécanisme mis à disposition par Java est l'exception.

Lorsque le système rencontre une situation anormale qui nécessite l'interruption du traitement en cours, il lance une exception. Cette exception va remonter la pile des appels en les interrompant jusqu'à être « catchée ».

Si personne ne « catch » l'exception celle-ci est affichée par le système en interrompant l'application. Attention ! dans le cadre d'un service REST, Spring REST « catche » toute exception non catchée en amont pour faire une réponse avec un statut http 500.

Le mécanisme d'exception permet :

- L'interruption du traitement,
- Le nommage de l'exception,
- Et l'affichage de sa pile.

Dans l'objectif de faciliter l'identification de l'origine du problème.

En conséquence,

- Si on lance une exception, il faut utiliser un nom d'exception qui soit significatif.
- Il ne faut surtout pas « avaler » une exception : catcher une exception sans renvoyer d'exception et sans tracer sa pile d'appel.

```
// Code à proscrire
try {

} catch (final NumberFormatException he) {
        // rien : cela ne devrait jamais arriver
}
```

A minima, il faut tracer l'exception :

```
// Code à préférer mais déconseillé
try {

} catch (final NumberFormatException he) {
        log.error("cela ne devrait jamais arriver",he);
}
```

Dans la pratique, il est déconseillé de « catcher » des runtime exception et encore plus `Exception` et, en aucun cas `Throwable` (`Throwable` est l'interface mère que partagent `Exception` et `Error`, « catcher » une Error peut potentiellement déstabiliser l'ensemble du serveur d'application)

Une runtime exception doit pouvoir être évitée par un code de vérification en amont comme la vérification qu'un objet n'est pas nul avant l'exécution de l'appel d'une méthode.

Les seules runtime exceptions qu'il est possible de catcher sont celle qui concernent les `IOException` : il n'est pas possible de savoir si un fichier est présent, lisible ou modifiable avant d'y accéder.

« `Catcher` » Throwable est formellement déconseillé, cela correspond à catcher des exceptions de la JVM comme le OutOfMemoryError pour laquelle rien ne peut être fait.

```
// Code à proscrire
try {

} catch (final Throwable he) {
        log.error("cela ne devrait jamais arriver",he);
}
```

Nous recommandons la création d'une exception mère pour les exceptions applicatives : on l'appellera `BusinessLogicException`. Elle héritera de la classe `Exception`.

L'exception doit aussi permettre de véhiculer les raisons de l'erreur vers l'appelant ou l'utilisateur. Il faut donc attacher une certaine importance aux propriétés de l'exception.

On recommandera de définir les propriétés suivantes pour la `BusinessLogicException` :

- errorCode : le code erreur, un code technique permettant d'identifier la cause de l'erreur
- origin : doit permettre de retrouver dans le code, l'erreur dans le code
- detail : une explication à destination de l'utilisateur.

Nous vous conseillons également de définir des exceptions plus précises qui héritent de la BusinessLogicException comme :

- ForbiddenException : si l'utilisateur n'est pas autorisé

- NotAllowedException : si l'utilisateur n'est pas authentifié
- NotFoundException : par exemple si on ne trouve pas un objet

Cette liste n'est pas exhaustive et peut être complétée : plus le nom de l'exception est précis plus le travail d'analyse est simplifié.

Si vous catchez une exception pour en envoyer une nouvelle, il faut faire en sorte que la pile de la première exception ne soit tracée qu'une et une seule fois.

On ne trace pas pour renvoyer :

```
// Code à proscrire : on risque de voir apparaître deux fois la pile de l'erreur
try {


} catch (final NumberFormatException he) {
        log.error("message",he);
        throw he;
}
```

On peut intercepter pour lancer sa propre exception mais dans ce cas, il faut passer l'exception interceptée en cause pour que le destinataire dispose de la pile complète.

```
// Code recommandé si on lance sa propre exception
try {


} catch (final NumberFormatException he) {
        throw new BusinessLogicException("mon message",he);
}


// Code à proscrire : on perd la pile de la première exception
try {


} catch (final NumberFormatException he) {
        throw new BusinessLogicException("mon message");
}
```

5.8.3 Service REST et Exception

En cas d'erreur, un service REST doit renvoyer des réponses http en utilisant les codes http disponibles.

La couche de transport de Spring REST intercepte toute erreur non interceptée en amont pour en faire une réponse http avec le code 500. Si on laisse faire, Spring REST par défaut, le message d'erreur renvoyé à l'utilisateur ne sera pas très précis et ne répondra pas à la norme des services REST.

Spring REST permet la définition d'un Handler d'exception coté Service REST chargé d'intercepter les erreurs et de retourner un message au client.

Il faut pour cela définir une classe « GlobalExceptionHandler » annotée avec ControllerAdvice.

Les méthodes de la classe sont annotées par ExceptionHandler en définissant l'exception qui sera interceptée. La méthode construit le message de réponse dans le cas de l'exception en précisant le statut http et le message d'erreur.

Il est recommandé de faire en sorte qu'il y ait des méthodes différentes pour chacune de vos exceptions métiers.

```
import static org.springframework.http.HttpStatus.INTERNAL_SERVER_ERROR;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {

   private static final Logger LOGGER = LogManager.getLogger(GlobalExceptionHandler.class);

   @ExceptionHandler({ RuntimeException.class })
   public ResponseEntity<String> handleRunTimeException(RuntimeException e) {
      return error(INTERNAL_SERVER_ERROR, e.getMessage());
   }

   private ResponseEntity<String> error(HttpStatus status, String message) {
      LOGGER.error(message);
      return ResponseEntity.status(status).body(message);
   }
}
```

Remarque : il ne s'agit que d'un exemple, d'autres exceptions peuvent être interceptées et la structure du message retournée peut l'être au format JSON.

Au niveau des contrôles métiers, il s'agit de soulever la bonne exception en précisant le message à retourner au client.



Catch dans le code d'un controller
Il est déconseillé de « catcher » une exception dans un controller. Le code du controller doit laisser l'exception remontée jusqu'à la classe GlobalExceptionHandler qui prendra en charge l'envoie de la réponse http appropriée.

5.9 GESTION DES LOG

5.9.1 OBJECTIF DE LA LOG

La log permet de tracer les actions du système d'information lors de son exécution : les solutions de log Java permettent de garder une trace du fonctionnement en production en enregistrement cette trace dans des fichiers.

L'exploitation peut utiliser ces informations pour surveiller le bon fonctionnement de l'application.

La log est particulièrement utile dans les phases d'installation et configuration d'une application : les informations qu'elle affiche doivent permettre de résoudre les problèmes de configuration.

La log permet également aux équipes de développement d'analyser les disfonctionnements détectés en production. Pour ce point, il est important que la pile d'une erreur soit présente dans la log mais que l'on est une idée des actions précédentes de l'utilisateur.

5.9.2 LOGGER JAVA

Les loggers Java permettent sur le principe de log4j2, de définir des niveaux de log que l'on pourra activer par configuration externe de la log.

Ils permettent d'avoir :

- une log horodatée,
- d'identifier le thread qui a émis la log : utile en production pour rattacher les différentes traces à un même utilisateur
- de définir un logger par classe : cette information peut être présente dans le fichier de log, ce qui permet de savoir qui a émis la log
- un mécanisme d'affichage d'une pile d'erreur

Tous ces éléments vont permettre à partir du fichier produit de retrouver le disfonctionnement et de reconstituer ce qui s'est passé.

Pour que cela soit possible, il ne faut pas casser le mécanisme en utilisant des mécanismes non gérer par le logger pour tracer des informations.



Ne jamais utiliser e.printStackTrace() pour tracer une exception
Ce mécanisme n'est pas gérer par le logger, et l'erreur peut ne pas être retrouvée dans le fichier de log.



Utiliser logger.error(« message », e) pour tracer l'erreur et la pile d'erreur.



Ne jamais utiliser System.out.println() même pour debugguer
Le logger dispose d'une méthode pour tracer en mode debug : logger.debug(« message »). Vous n'aurez pas besoin de supprimer la ligne avant de livrer et vous ne risquez pas de l'oublier.

Utiliser un logger par classe

```
private static final Logger logger =
LoggerFactory.getLogger(this.getClass());
```

En définissant un logger par classe, on facilite l'identification de la ligne émettrice de la log.

Ne pas faire de traitement conditionnel en fonction du niveau de log

Il faut à tout prix éviter de faire des actions uniquement sur condition du niveau de log. Cela crée de la complexité cyclomatique et rend le comportement de l'application différent en fonction du niveau de trace (heisenbug).

Utiliser des lambdas pour tracer les niveaux « debug » et « trace », ou dans le cas de traces structures complexes pour les niveaux moins fins.

La trace d'informations peut être coûteuse en temps. Malheureusement, l'évaluation de ce qui est à tracer se fait dans tous les cas, même si le niveau de trace ne permet pas la sortie du message, on fait donc du traitement pour rien.

Depuis Java 8, il est possible de passer des callbacks aux services de log, ce qui permet de différer l'évaluation du contenu à tracer, au moment où le framework de trace sait s'il doit laisser sortir le message ou non.

L'appel se fait de cette manière :

LOGGER.debug(() -> « ma trace avec cet objet coûteux à évaluer : {} », obj) ;

5.9.3 DEFINITION DES NIVEAUX DE LOG

Les loggers Java permettent la définition de 6 niveaux hiérarchisés de log :

- Trace : le niveau le plus bas
- Debug : niveau utilisé pour le debugging.
- Info
- Warn
- Error
- Fatal

La configuration du logger permet de définir à partir de quel niveau on trace l'information dans le fichier, réduisant le coût des logs de niveau inférieur. Si on définit qu'en production on trace à partir du niveau info, le coût des logs

de niveau Debug ou Trace sera négligeable et peut rester dans le code si on respecte les recommandations précédentes.

Le choix d'un niveau de log dépend principalement du choix de leur présence en production ou non. Pour celle présente en production, cela dépend également du mode de travail de l'équipe d'exploitation et des conventions définies entre le développement et l'exploitation.

De manière générale, les équipes d'exploitation sont toutes habituées à suivre les ERROR et les FATAL : elles considèrent que ce qui est en error ou fatal nécessite une action de leur part. On évitera l'utilisation de ces niveaux de log pour tracer une erreur de l'utilisateur comme une saisie invalide.

Pour les WARN cela dépend des équipes d'exploitation. Certaines sont attentives à son occurrence d'autre non. Dans le cadre de l'ABM, il est convenu que ce niveau sera utilisé pour tracer les erreurs utilisateurs.

Si nous nous limitons en production à ces 3 niveaux (WARN, ERROR, FATAL), nous n'aurons qu'une trace de la survenue d'un incident. Il ne sera pas possible de le recontextualiser dans un flux d'actions. La possibilité de retrouver dans quel contexte, quelle série d'actions à amener l'incident, améliore grandement le suivi de l'exploitation.

Le trafic des applications de gestion comme celles de l'Agence et l'efficacité du système de log Java permet d'envisager de tracer de 1 à 3 lignes par actions utilisateur en production.

Nous conseillons de positionner le seuil de log pour la production au niveau INFO et d'utiliser ce niveau pour tracer au moins une ligne INFO par action utilisateur.

La ligne tracée en INFO doit être si possible, une synthèse du traitement réalisé par le service REST concerné : idéalement s'il s'agit d'une création ou mise à jour en base, il faudrait tracer le login de l'utilisateur et l'identifiant de l'objet créé ou mis à jour.

On n'utilisera pas le niveau INFO pour tracer l'entrée dans le service REST (à ce niveau, il est rarement possible de dire si tout s'est bien passé). Si on veut garder une trace de l'entrée dans un service REST, nous conseillons le niveau DEBUG.

Le niveau DEBUG est généralement utilisé comme son nom l'indique pour le DEBUG.

En synthèse, on peut retenir les règles suivantes :



On utilisera les 4 niveaux de log suivant : DEBUG, INFO, WARN, ERROR

DEBUG sera utilisé pour le debug ou garder une trace d'entrée dans un service.

INFO sera utilisé pour garder une trace de succès des actions utilisateur importantes. On essaiera de tracer le login de l'utilisateur et l'identifiant de l'objet impacté.

WARN sera utilisé pour tracer les erreurs utilisateurs qu'il doit être en capacité de résoudre lui-même.

ERROR sera utilisé pour toutes les erreurs techniques



Le logger sera configuré en production pour afficher les niveaux INFO et supérieurs.



Un service REST devra tracer en cas de succès, au minimum une ligne en INFO. Cette ligne devra être une synthèse de l'opération réalisée.

La ligne pourra contenir le login de l'utilisateur et l'identifiant de l'objet impacté

5.9.4 MESSAGE DE LOG

L'analyse d'une log doit permettre de retrouver le cheminent du code : c'est pour cette raison qu'il est utile d'avoir dans certain cas, une trace des traitements réussis en amont.

Toutefois, pour être en mesure de reconstituer ce cheminement, il est impératif de pouvoir lier la ligne de log avec une ligne de code.

Pour que cela soit simple et facile, nous recommandons que chaque ligne de log commence par le nom de la méthode qui contient l'instruction logger. Cette règle doit être appliquée dans tous les cas du DEBUG à ERROR.



Une ligne de log doit toujours commencer par le nom de la méthode qui l'a émise.

Cette règle doit être respectée y compris pour le DEBUG et ERROR .

Un message de log doit être unique : l'utilisation du nom de la méthode aide à l'unicité, mais il faut éviter les messages qui n'affichent qu'une valeur, il faut au moins préciser la variable concernée.

Les messages de niveau INFO, WARN et ERROR sont destinés à une exploitation en production, ils doivent être clairs et significatifs en indiquant l'objet principale (par son identifiant par exemple) et son contexte en précisant par exemple la valeur erronée.

Pour les messages en INFO, des valeurs significatives pour le métier peuvent être ajoutées à l'identifiant. Attention à toutefois ne pas divulguer d'informations sensibles dans la log.

Pensez qu'une des façons principales d'utilisation de la log est la recherche d'une chaine de caractères : essayez de faciliter cette opération.



Nous vous déconseillons l'utilisation des mots ERROR et FATAL dans vos messages
Ces mots peuvent être utilisés par l'équipe d'exploitation pour détecter les éventuels disfonctionnements. Il n'est pas judicieux que cela ramène une ligne indiquant une erreur de saisie d'un utilisateur. Cela crée des faux positifs qui rendent l'exploitation délicate.

Le message de log sera structuré d'une telle manière qu'il sera facile à lire et à retrouver :

- Le message commencera toujours par le nom de la méthode
- Une valeur sera toujours préfixée par le nom de la variable la contenant ou du nom du libellé dans l'écran : il est plus facile et pertinent de rechercher « Donneur:200900 » que « 200900 ». La seconde peut ramener également un Utilisateur 200900, un patient 200900…

Toujours préfixer une valeur par son nom.

Cela permet de retrouver plus facilement les lignes où l'objet a été manipulé et de reconstituer son évolution.

Il est important pour le suivi d'exploitation de garder une trace des cas les plus imprévisibles. En conséquence, dans tous les blocs catch, il est nécessaire a minima de tracer l'erreur détectée.

Tout bloc catch d'une RuntimeException doit tracer une ERROR.

`Les seuls blocs catch qui ne sont pas soumis à cette règle sont ceux qui intercepte une exception fonctionnelle héritant de BusinessLogicException.`

`Attention ! si on fait dans le catch un throw, il ne faut pas logger la pile, uniquement le message.`

Pour la reconstitution du cheminent d'un traitement, il est important de conserver une trace du cas fonctionnel dans lequel se range une action particulièrement si une partie du traitement est spécifique (par exemple si le patient est mineur). Dans ce cas, il faudra une ligne de log de niveau INFO précisant le passage dans la branche concernée.

Si en debug, vous souhaitez tracer un ensemble d'informations couteuses, il existe un test « if(logger.isDebugEnabled()) » avant le bloc contenant logger.debug().

L'utilisation du logger.isDebugEnabled est à proscrire
Ce type de bloc conditionnel en fonction du niveau de log est dangereux : il peut masquer en développement des comportements erronés en production : souvent si le bloc est couteux c'est qu'il modifie l'état du système (initialise une valeur, charge un enregistrement…).

5.10 DEFINITION D'UN CONTROLLER

5.10.1 LE CONTROLLER

Le controller est une classe annotée par RestController pour ResponseEntityController. Il s'agit d'une annotation du framework spring-web qui est le framework retenu pour définir un service REST.

Deux autres annotations de la classe complète la définition du controller :

- CrossOrigin pour autoriser les requêtes CORS
- RequestMapping qui permet de définir le path de la ressource

On définira un controller par ressource.



Le Controller est défini pour une ressource et doit être annoté par les 3 annotation suivantes

```
@CrossOrigin
@RestController
@RequestMapping("/nom-ressources")
public class NomRessourceRestController {
```

Où le path est le nom de la ressource en français et au pluriel en utilisant des minuscules et le tiret comme séparateur de mot

Le controller est ses méthodes ont en charge la partie communication du service. Dans la répartition des rôles l'accès à la base de données et de la responsabilité du dataservice.

En conséquence, il n'est pas judicieux d'annoter une méthode du controller avec `Transactional`. Cela ne sert qu'à masquer une mauvaise gestion du chargement des données au niveau du dataservice.

L'annotation Transactional est souvent ajoutée à une méthode du controller pour éviter l'apparition d'une exception de « lazy loading » lorsque le mapper construit le DTO à partir d'une entité JPA dont les objets liés n'ont pas chargé.



Nous vous déconseillons l'annotation d'une méthode du controller avec @Transactional
Il est préférable soit de charger les objets liés manquant au niveau du DataService soit de modifier le mapper ou le DTO pour ne pas renvoyer l'objet lié.
La seconde approche est souvent à préférer car l'utilisation du Transactional sert à masquer des problèmes de chargement d'entités liées à des 3ieme ou 4ieme niveau qui ne devraient pas être renvoyées au front.

Si vous avez besoin d'accéder à plusieurs DataService pour réaliser le traitement métier attendu, ce sera le premier dataservice qui aura la charge d'invoquer les autres et non pas le controller.

5.10.2 LES METHODES D'UN CONTROLLER

Le controller définit pour la ressource, les différents endpoints et les méthodes http permettant leur accès.



Il faut utiliser les méthodes http pour définir les actions sur la ressource et non pas des verbes dans l'URI du endpoint.

Pour récupérer une instance de la ressource, il faut utiliser la méthode GET en précisant l'id de la ressource comme `PathVariable`.

L'annotation `GetMapping` permet de définir la méthode GET et la `PathVariable`

```
@GetMapping(path = "/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
```

```
public EvenementIndesirableDTO getEvenementIndesirable(@RequestHeader MultiValueMap<String, String>
headers,@PathVariable("id") Long id) throws NotFoundException  {
```



La récupération d'une instance d'une ressource se fait avec un GET et son id comme PathVariable.

```
@GetMapping(path= « /{id} »
```

Où le pathVariable est récupéré comme paramètre de la méthode avec l'annotation @PathVariable(« id »)

Pour récupérer une liste d'instances de la ressource, il faut utiliser la méthode GET en précisant les éventuels critères de recherche en Query string.



Pour les paramètres de la query string, on utilise lowerCamelCase : en minuscule avec des majuscules.

Les paramètres nécessaires à la pagination sont également passés dans la query string.



Pour la pagination, les paramètres de la query string seront :

size : nombre maximum d'enregistrements retournés dans une page

page : numéro de la page en prenant comme première page 0

sort : le tri à prendre en considération. La direction peut être précisée avec (asc) ou (desc).

Pour récupérer une liste d'instances, il n'est pas nécessaire de préciser l'URI du endpoint.

```
@GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
public List<EvenementIndesirableDTO> getListEvenementIndesirable(@RequestHeader MultiValueMap<String,
String> headers,
@RequestParam(name = "size", required = true) int size,
@RequestParam(name = "page", required = true) int page,
@RequestParam(name = "sort", required = false) String sort
```



La récupération d'une liste d'instances doit renvoyer le statut http :

200 OK si la liste renvoyée est complète (pas de pagination),

206 Partial Content si la liste récupérée n'est qu'une partie des enregistrements vérifiant les critères dans le cas d'une pagination.

La création d'une instance doit utiliser la méthode http POST et passer l'entité dans le body de la requête sous forme de JSON.

Sprint REST met à disposition l'annotation `PostMapping`.

Il n'est pas conseillé de préciser le path.

La méthode `ResponseStatus` permet de préciser le statut http si le traitement se passe bien. Pour la création, il sera précisé pour renvoyer le code 201 au lieu du défaut (code 200).



La création d'une instance doit utiliser le POST est renvoyé un 201 Created en cas de succès.

```
@ResponseStatus(code=HttpStatus.CREATED)
@PostMapping(…
```

La mise à jour d'une ressource ne concerne généralement qu'une partie de la ressource. Nous conseillons l'utilisation de la méthode http `PATCH` pour les mises à jour car elle permet une mise à jour partielle qui est plus adaptée surtout pour le cas de ressource ayant un grand nombre de propriétés (supérieur à 20).

La mise à jour partielle permet d'éviter d'écraser une mise à jour concurrente qui aurait été faite par un autre utilisateur : si par exemple un premier utilisateur met à jour les informations médicales d'une personne pendant qu'un autre met à jour son adresse.

En l'absence de verrouillage de la ressource personne, les deux utilisateurs vont lire l'instance, la personne qui a mis à jour les données médicales soumet son enregistrement en premier. L'instance en base prend donc ses modifications. Quand le second utilisateur enregistre, ses données sont celles de départ, les données médicales sont donc celles d'origine. Si on utilise une mise à jour complète avec le `PUT`, il va envoyer les données qu'il a mis à jour (l'adresse) mais aussi toutes les autres données de la ressource (en l'occurrence les données médicales) même celles qu'il n'a pas modifié.

Le résultat du `PUT` est l'écrasement des données médicales mises à jour par le premier utilisateur par les données d'origine renvoyées par le second utilisateur.

Le PATCH permet au premier utilisateur de n'envoyer que les données médicales et au second l'adresse. Si on ne met à jour que les données reçues, on n'écrase pas la mise à jour de l'autre utilisateur.

Un service `PATCH` prend dans l'URI du endpoint, la valeur de l'id (en pathvariable) de la ressource qui doit être mise à jour.

Il est possible de définir plusieurs services de mise à jour : un par partie mise à jour (dans notre exemple un pour l'adresse et un pour les données médicales).

La mise à jour d'une instance doit préférer le PATCH en ne passant qu'une partie des données qui pourront être préciser par le path.

```
@PatchMapping(path= «/{id}/ma-partie »
```

Où le pathVariable est récupéré comme paramètre de la méthode avec l'annotation @PathVariable(« id »)

Le body peut prendre un DTO qui correspond à une partie de la ressource

Le statut renvoyé en cas de succès est 200 OK.

Certaines mises à jour correspondent à un changement de statut ou d'état d'un enregistrement comme par exemple une annulation ou une validation/approbation.

Dans ce cas précis, il est possible d'utiliser un verbe pour préciser l'uri avec le nom de l'action. On utilisera dans ce cas la méthode `PATCH` qui ne prendra probablement aucune information dans le body de la requête.

Pour changer l'état ou le statut d'une instance il est possible d'utiliser le PATCH en précisant l'action par un verbe.

```
@PatchMapping(path= «/{id}/verbe »
```

Où le pathVariable est récupéré comme paramètre de la méthode avec l'annotation @PathVariable(« id »)

Le body peut être vide.

Le statut renvoyé en cas de succès est 200 OK.

Le service `DELETE` pour la suppression d'une ressource est rarement utilisé dans les applications de gestion, si besoin on utilisera la méthode http DELETE en précisant comme pathvariable l'id de la ressource.

La suppression d'une instance doit utiliser le DELETE.

```
@DeleteMapping(path= «/{id}»
```

Où le pathVariable est récupéré comme paramètre de la méthode avec l'annotation @PathVariable(« id »)

Le statut renvoyé en cas de succès est 204 No Content.

Les méthodes POST, PUT et PATCH doivent renvoyer dans le body de la réponse, l'instance mise à jour.

5.10.3 IDENTIFICATION DE L'UTILISATEUR

Certaines méthodes doivent vérifier que l'utilisateur dispose des droits, qu'il a le bon profil ou tracer le nom de l'utilisateur pour des raisons d'audit.

La méthode ne doit pas faire confiance à l'information envoyée par le front dans un paramètre. Elle doit récupérer le login de l'utilisateur qui a été positionné par l'API Manager dans le header X-USER-LOGIN et utiliser cette information pour vérifier ces droits.



L'identification de l'utilisateur ne doit se faire que par l'utilisation du header X-USER-LOGIN positionné par l'API Manager.

Le Front peut envoyer des informations qui précisent les droits de l'utilisateur quand ce dernier peut utiliser différents profils comme :

- L'établissement auquel il est rattaché
- Sa profession

Cette information doit être envoyée dans le header de la requête.

5.11 DEFINITION D'UN DATASERVICE

5.11.1 LE DATASERVICE

Le dataservice est une classe annotée par @Service. Cela permet d'indiquer qu'il s'agit d'une classe qui est en charge de coder les règles métier



Le DataService est annoté par @Service.

5.11.2 METHODE ANNOTEE @TRANSACTIONAL ET ACCES BASE

Le DataService est une classe qui utilise les interfaces Repository pour l'implémentation des règles de gestion métier.

La première méthode du DataService appelée par le controller doit ouvrir la transaction base afin que tous les accès base réalisés par la suite soit dans la même session hibernate. Pour cela, il faut annoter la méthode avec `@Transactional.`

Il est important que tous les échanges avec la base soient dans une même et unique session hibernate, ce qui est assuré par l'ouverture d'une transaction :

- cela permet de récupérer un objet et de l'utiliser dans un autre accès, par exemple, lors d'une mise à jour : dans un premier temps on récupère l'objet et dans un second temps on le met à jour.
- Si on dans une session, il est possible de parcourir les liens ManyToOne entre deux instances.



La première méthode d'un DataService appelé par le controller doit être annotée par @Transactional.

`Il s'agit de l'annotation Transaction de Spring. Le mode de propagation de la transaction doit être REQUIRED qui est le mode par défaut.`

La méthode du DataService doit s'assurer que toutes les entités liées qui doivent être renvoyées par le DataService, aient été chargées par la méthode : une fois la méthode terminée, la transaction sera fermée et il ne sera plus possible de charger une entité lorsqu'on parcours un lien entre deux entités JPA.

Un DataService ne doit accéder à la base qu'à travers l'utilisation de Repository. Cela garantit que seule une syntaxe portable de l'accès à la base de données sera présente dans le code.



Un DataService ne doit utiliser que les interfaces Repository pour accéder à la base.



L'usage de jdbc pour accéder à la base de données et à proscrire.

5.12 DEFINITION D'UN REPOSITORY

5.12.1 LE REPOSITORY

Le Repository est une interface annotée par @Repository qui hérite de la classe `JpaRepository`.

Cette interface permet de définir les accès à la base de données et dispose de méthodes prêtes à l'emploi comme des méthodes permettant de gérer la pagination.

5.12.2 OPTIMISATION DES RECHERCHES

Le Repository sert principalement à définir des requêtes de recherche d'informations.

Dans la grande majorité des cas, un objet est lié à d'autres objets et plusieurs d'entre eux sont nécessaires au front : il va falloir les charger.

Pour optimiser les temps de réponse, il est souhaitable de charger en une seule requête les différents objets. Pour cela, il est possible d'annoter la méthode avec `EntityGraph`.

Cette annotation permet grâce à son attribut attributePath, la définition de la liste des entités liées à charger.

5.12.3 UTILISATION DU FINDBYID

La méthode findById permet la récupération d'un objet à partir de son id. Cette méthode renvoie un Optional. Un Optional dispose d'une méthode isPresent() pour savoir si un objet a été trouvé et d'une méthode get() pour récupérer l'objet trouvé.

Il est impératif de stocker temporairement dans une variable Optional le résultat d'un findById avant de le tester ou de récupérer la valeur.

```
// Code à proscrire : provoque deux SELECT en base
if(acteurConcerneRepository.findById(acteurConcernePK).isPresent()) {
ActeurConcerne acteurConcerne = acteurConcerneRepository.findById(acteurConcernePK).get();
```

```
// Code à préférer
Optional<ActeurConcerne> opt= acteurConcerneRepository.findById(acteurConcernePK);
if(opt.isPresent()) {
ActeurConcerne acteurConcerne opt.get();
```

La méthode findById doit être préférée à la méthode findOne : les deux méthodes retournent une instance à partir d'un identifiant, mais la méthode findOne ne valorise pas les propriétés de l'instance.

Nous déconseillons l'utilisation de la méthode findOne
Préférer la méthode findById.

La méthode findById peut être associée avec l'annotation EntityGraph afin de charger en une seule fois les entités liées.

5.13 DEFINITION D'UNE ENTITE

5.13.1 ENTITE JPA

Une entité JPA est une classe Java permettant de faire le lien entre la classe Java et le modèle physique de la base.

Une classe Java doit posséder une et une seule propriété annotée Id ou EmbeddedId pour une clé composée. Cette propriété doit impérativement correspondre à un index unique comme une clé primaire. JPA/Hibernate ne fonctionne pas s'il récupère deux enregistrements ou plus pour une recherche avec l'id.

Une entité JPA doit posséder un Id ou EmbeddedId correspond à un index unique.

Une entité JPA doit être annoté avec

- Entity
- Table afin de préciser le nom de la table ou de la vue.

5.13.2 DEFINITION DES PROPRIETES

Les propriétés simples d'une entité peuvent être annotées avec Column afin de définir le nom de la colonne.



Nous conseillons d'annoter toutes les propriétés avec Column.

Cela permet l'utilisation de nom de propriétés correspondant à un concept métier et non pas au nom de la colonne.

5.13.3 DYNAMICUPDATE

Par défaut, la mise à jour d'une entité JPA est complète : toutes les colonnes sont positionnées qu'elles aient été modifiées ou non.

Pour éviter qu'une mise à jour, écrase avec de vieilles valeurs des données qui auraient été modifiées récemment, il est souhaitable d'annoter la classe avec DynamicUpdate : une annotation hibernate qui permet de ne mettre à jour que les propriétés qui ont été modifiées depuis la lecture en base.

5.14 RECHERCHE D'ENREGISTREMENT

5.14.1 OBJECTIF

L'objectif est d'éviter de ramener coté Frontend tous les enregistrements recherchés.

A cet effet, chaque recherche comportera l'index de la page à afficher, le nombre d'enregistrements à récupérer, l'ordre de tri à appliquer.

Elle devra également être complétée par l'appel à un service chargé de ramener le nombre d'enregistrements.

5.14.2 IMPLEMENTATION COTE SERVICE REST

La signature d'une méthode de recherche comportera les attributs suivants :

- size : nombre d'enregistrement à retourner.
- page : index à partir duquel retourner les enregistrements. On commence à 0.
- sort : ordre de tri sous la forme : champ1(asc),champ2(desc)

Exemple d'appel au service :

```
http://mon-serveur/nom-ressources?size=20&page=0&sort_order=champ1(asc),champ2(desc)
```

Exemple de signature de la méthode appelée :

```
/**
 * Paginated and sorted search
 */
@GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
public List<Message> getAll (@RequestHeader MultiValueMap<String, String> headers,
```

```
@RequestParam(name = "size", required = false) int limit,
@RequestParam(name = "page", required = false) int offset,
@RequestParam(name = "sort", required = false) String sortOrder)  {
```

5.14.3 IMPLEMENTATION COTE DATASERVICE

Le dataservice se contente de récupérer les paramètres et d'appeler les méthodes du Repository chargées de les appliquer.

```
/**
 * Paginated and sorted search
*/
public List<DossierSignalement> getAllDossierSignalements(int page,int size, String sortOrder) {
  final Pageable pageable = PageRequest.of(page, size, getSortOrder(sortOrder));
  return dossierSignalementRepository.findAll(pageable).getContent();
}
```

5.14.4 IMPLEMENTATION COTE REPOSITORY

Exemple d'implémentation :

```
String SELECT_CLAUSE = "select ds ";
String SQL_CLAUSE = "from DossierSignalement  ds, Donneur do " +
                    "where ds.numeroCristal=do.numeroCristal";
@Override
@Query(value = SELECT_CLAUSE + SQL_CLAUSE)
Page<DossierSignalement> findAll(Pageable pageable);
```

5.14.5 RECHERCHE DU NOMBRE D'ENREGISTREMENTS TROUVES

La recherche du nombre d'enregistrements trouvé s'effectue de la façon suivante :

```
 @Query(value = SELECT_COUNT_CLAUSE + SQL_CLAUSE)
Long countAll();
```

6 Développement Frontend

6.1 AVANT PROPOS

Langage utilisé

Le français sera utilisé pour nommer les objets, leurs attributs ou leurs méthodes. Le but est d'éviter toute interprétation.

6.1.1 APPROCHE GENERALE : LE FRAMEWORK ANGULAR

Angular est un framework pour le langage Javascript permettant le développement d'applications web. C'est un framework qui met à disposition des libraires qui vont faciliter le développement des applications frontend.

Ce framework est orienté « composant » c'est-à-dire qu'il va isoler le HTML, le CSS et la logique d'un composant (typescript) afin que ce dernier soit indépendant et qu'il soit réutilisable, on parle alors de « component API ».

Enfin, Angular étant basé sur Javascript, il va permettre à l'application d'être dynamique. L'application va pouvoir réaliser des opérations de CRUD vers un serveur et également interagir aux actions de l'utilisateur. Pour rendre tout cela possible, Angular s'appuie sur :

- Directive : leur rôle est de donner des instructions au DOM HTML afin de rendre ce dernier dynamique. Trois types de directives :
  - Directives structurelles : elles vont ajouter ou enlever des éléments du DOM. Elles agissent donc directement sur la structure du DOM.
  - Directives de composant : elles vont permettre via un sélecteur d'ajouter un composant dans un autre.
  - Directives d'attributs : elles vont permettre de modifier les comportement et l'apparence d'un élément du DOM. Elles n'agissent pas sur la structure du DOM.
- Les services : avec le module HttpClient mise à disposition par Angular qui s'appuie sur l'API XmlHttpRequest, l'application sera en mesure des réaliser des requêtes http vers un serveur.
- Pipe : ce sont des filtres qui vont afficher des données reformatées sans modifier la valeur de la source.
- Guards : interfaces qui vont permettre de protéger les ressources d'une application.
- Interceptor : interface qui va permettre de d'intercepter les requêtes http.

6.2 CONFIGURATION DU PROJET

6.2.1 PILE TECHNIQUE

Tous les développeurs auront le même environnement technique pour le développement de l'application :

- Angular 11.2.10
- PrimeNg 11.
- IntelliJ IDEA (x64 bit)
- NodeJS 12.
- Maven
- JDK 11

- JBOSS 7.3.6 EAP

Il est à noter que l'éditeur IntelliJ IDEA permet l'écriture de code de type Typescript (qui est le langage sur lequel est basé Angular) et de Java.

6.2.2 INSTALLATION DU CLIENT ANGULAR

Afin que tous les développeurs puissent travailler avec la même version d'Angular, il est nécessaire d'installer en global la version 11.2.10 qui a été choisis pour le projet (dernière version LTS pour Avril 2011). Trois commandes à réaliser pour s'en assurer :

- npm uninstall -g @angular/cli
- npm install -g @angular/cli@11.2.10
- npm install

6.2.3 GESTION DES DEPENDANCES

Le développement de la partie Frontend s'appuie sur les modules et librairies suivantes :

```
"dependencies": {
  "@angular/animations": "11.2.10",
  "@angular/cdk": "11.2.9",
  "@angular/common": "11.2.10",
  "@angular/compiler": "11.2.10",
  "@angular/core": "11.2.10",
  "@angular/forms": "11.2.10",
  "@angular/platform-browser": "11.2.10",
  "@angular/platform-browser-dynamic": "11.2.10",
  "@angular/router": "11.2.10",
  "@fullcalendar/core": "4.0.2",
  "@fullcalendar/daygrid": "4.0.1",
  "@fullcalendar/interaction": "4.0.2",
  "@fullcalendar/timegrid": "4.0.1",
  "@ngrx/data": "11.1.0",
  "@ngrx/effects": "11.1.0",
  "@ngrx/entity": "11.1.0",
  "@ngrx/router-store": "11.1.0",
  "@ngrx/store": "11.1.0",
  "@nrwl/angular": "12.0.7",
  "@rxweb/reactive-form-validators": "2.1.2",
  "@types/jsdom": "16.2.1",
  "@types/lodash-es": "4.17.5",
  "chart.js": "2.9.4",
  "chartjs-plugin-datalabels": "0.7.0",
```

```
  "date-fns": "2.22.1",
  "jwt-decode": "3.1.2",
  "keycloak-angular": "8.4.0",
  "keycloak-js": "12.0.4",
  "lodash-es": "4.17.15",
  "moment": "2.24.0",
  "ng-let": "2.0.1",
  "ngrx-store-localstorage": "11.0.0",
  "ngx-logger": "4.3.3",
  "ngx-sse-client": "2.0.0",
  "primeflex": "2.0.0",
  "primeicons": "4.1.0",
  "primeng": "11.3.2",
  "quill": "1.3.7",
  "roboto-fontface": "0.10.0",
  "rxjs": "6.5.5",
  "tslib": "2.0.0",
  "web-animations-js": "2.3.2",
  "zone.js": "0.10.2"
 }
```

6.2.4 GESTION DES VERSIONS

La gestion des versions est effectuée via SVN comme pour la partie Backend.

Le nom du projet reflètera le domaine fonctionnel qu'il couvre : exemple, **« syrenad-front ».**

6.3 STRUCTURE DE L'APPLICATION

6.3.1 STRUCTURE APPLICATION WEB ANGULAR

Une application web Angular est dite SPA pour « Single Page Application ». Ainsi, l'application est un ensemble de composants regroupés les uns avec les autres qui vont former la page. Nous avons en effet qu'une seule page qui est **index.html**, chargée dès le départ, qui est le point d'entrée de l'application et ensuite nous auront le chargement des autres composants au besoin, selon les actions de l'utilisateur, qui viendront remplir de façon dynamique la page principale index.html.

En effet, c'est ensuite l'objet **Router** propre à Angular qui va alors gérer le chargement et l'affichage dynamique des composants.

6.3.2 STRUCTURE DU PROJET

Le répertoire app du répertoire src d'un projet Angular contient les sources applicatives.

Il est décomposé en 5 sous répertoires

- core : module toujours chargé qui contient des ressources toujours présentes pour l'application. Dans la pratique, seul l'architecte du projet peut modifier son contenu.
- features : module regroupant les fonctionnalités du projet
- public : contient les fonctionnalités accessibles sans authentification. En général, cela se limite à la partie authentification.
- shared : les ressources partagées entre les différents modules du répertoire features.
- store : répertoire pour la gestion du store (voir plus loin pour une explication du store)

Le répertoire shared peut être décomposé de deux manières différentes suivant la taille du projet.

Pour un projet de petite taille :

- Directives
- Fragments
- Layout
- Pipes
- Utilities

Chaque répertoire contenant les ressources de son type qui sont partagées entre les différents modules.

Pour un projet de taille plus importante, les ressources partagées peuvent être trop importante et nécessiter une plus grande structuration.

Dans ce cas, l'organisation suivante est recommandée :

- N répertoires de features partagées : chaque sous répertoire contient des ressources partagées organisées par domaine fonctionnel
- generic : les ressources communes non rattachées à un domaine fonctionnel.
- mock : pour avoir un mode bouchon.

Les répertoires generic et de features partagées sont organisés avec les sous répertoires suivant :

- directives
- enumerations
- fragments
- models
- pipes
- utilities

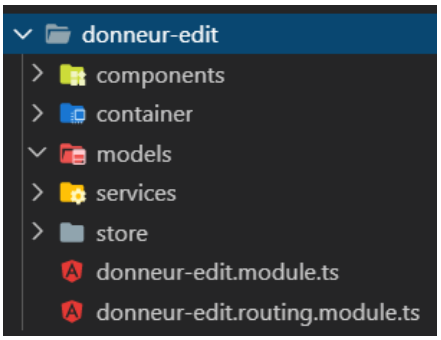
6.3.3 ARCHITECTURE DES MODULES FONCTIONNELS

Le répertoire features est décomposé en sous répertoires par domaine fonctionnel : un domaine fonctionnel peut correspondre à la gestion d'une ressource REST. Cette organisation permet un regroupement logique des ressources.

Chacun de ses sous répertoires sera décomposé de la manière suivante :

- guards
- models : contient les objets correspondant au modèle de données : les dto ou classe métier
- services : les services dont les services d'accès aux ressources REST du backend.
- store : gestion du store propre à la feature
- fragments : les components Angular qui gère l'affichage d'un morceau de page.
- containers ou pages : les components qui correspondent à l'équivalent fonctionnel de la zone principale de la page HTML. Elle est composée de fragments. Le terme containers sera utilisé pour les architecture qui utilise le store et pages les autres.

Ces modules fonctionnels respecteront l'architecture suivantes :



Chaque module fonctionnel aura son propre routing afin de gérer le chargement à la demande (lazy loading).

6.3.4 STORE

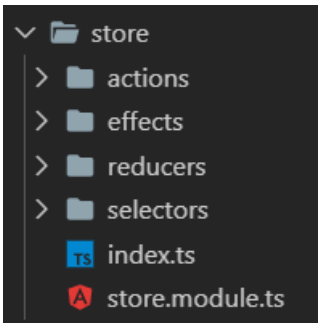
Le store sera utilisé dans l'application afin de gérer l'état de l'application. Le store va s'appuyer sur la librairie NgRX. Cette libraire s'appuie sur le pattern Redux qui définis trois grands principes :

- Unique source de vérité : Un seul état de l'application, le **Store**.
- L'état est en lecture seule : les propriétés du store ne peuvent pas être modifiés et seuls des **actions** vont mettre à jour l'état.
- L'état ne peut être modifié que par des fonctions pures, les **Reducers**.

Les commandes pour installer le store et profiter des principales fonctionnalités :

- npm i @ngrx/store
- npm i @ngrx/effects
- npm i @ngrx/router-store
- npm i @ngrx/store-devtools

Au niveau de l'architecture physique, le store se présente comme ceci :



- actions : permet de définir les actions que nous allons dispatcher.
- reducers : les fonction pures qui vont écouter les actions afin de modifier le state.
- effect : gère les actions de type asynchrone.
- selectors : permet de sélectionner des propriété du store.

Enfin, la configuration nécessaire du module :

```
import { NgModule } from '@angular/core';
import { StoreModule } from '@ngrx/store';
import { ReducerApp } from '.';
import { StoreDevtoolsModule } from '@ngrx/store-devtools';
import {EffectsModule } from '@ngrx/effects';
import { TodoEffects } from './effects/todo.effects';


@NgModule({
  declarations: [],
  imports: [
    StoreModule.forRoot(ReducerApp),
    EffectsModule.forRoot([TodoEffects]),
    StoreDevtoolsModule.instrument({
      name: ' [ TODO ] ',
      maxAge: 25
    })
  ]
})
export class TodoStoreModule { }
```

6.4 BONNES PRATIQUES

6.4.1 CONVENTION DE NOMMAGE

Le nommage des classes que les développeurs devront tous respecter seront les suivants où xxxx est le nom de base du composant:

- xxxx-fg.component.ts : pour les composants fragment
- xxxx-container.component.ts : pour les composants container
- xxxx.service.ts : pour les services
- xxxx.guard.ts : pour les guard
- xxxx.interceptor.ts : pour les interceptors
- xxxx.directive.ts : pour les directives
- xxxx.pipe.ts : pour les pipes
- xxxx.model.ts : pour les model

6.4.2 TYPAGE

Typescript est une surcouche du Javascript qui permet d'avoir une syntaxe plus claire pour le développement d'application web avec notamment les notions de classe, d'interface, de constructeur etc…

Et un des aspects les plus important du Typescript est le typage des variables. C'est pourquoi les développeurs devront respecter le typage des variables, ce qui n'est pas le cas en Javascript.

L'une des raisons est la maintenabilité du code. En effet, lorsque les variables sont typées, la compréhension et la lecture du code devient plus simple, logique et cohérente. La cohérence notamment avec les dto implémentés coté front qui vont permettre de retourner le type attendu lors d'une requête vers le serveur.

6.4.3 PATTERN « SEPARATION DES PREOCCUPATIONS »

C'est un principe de conception visant à segmenter un programme informatique en plusieurs parties, afin que chacune d'entre elles isole et gère un aspect précis de la problématique générale1. C'est une bonne pratique largement utilisée dans le développement.

Coté front ceci va se traduire par des modules qui vont traiter une partie de l'application. Une fois les modules définis, ils seront chargés à la demande, répondant au besoin de l'utilisateur.

C'est pourquoi nous disons qu'une application Angular est dite modulaire. A savoir découpée en module qui répond à un besoin spécifique et dont on va charger afin d'utiliser ses fonctionnalités.

Parmi le type de module que nous allons retrouver dans l'application nous retrouverons :

- Les modules Features : Ces modules qui seront chargés à la demande regrouperont les composants propres à la feature et chargés à la demande en fonction des actions des utilisateurs.
- Core Module : Ce module chargé dès le départ de l'application et met à disposition pour tous les composants des fonctionnalités nécessaires tels que les Pipes, Guard, Inteceptors, Enum…
- Shared Module : Ce module importé par les autres modules, au besoin, regroupe des composants qui seront partagés par plusieurs composants.
- Les modules de Routing : Ces modules permettront de gérer la navigation.

6.4.4 COMPOSANT TYPE ANGULAR

Un composant Angular type est définis par 4 fichiers que sont :

- Un fichier Typescript : on y retrouve toute la logique, des fonctions, variables
- Un fichier HTML : représente le rendu visuel du composant avec les données provenant de son fichier Typescript.
- Un fichier CSS : fichier qui représente le CSS du composant.
- Un fichier spec : fichier de test.



Remarque

Angular a choisi l'émulation du ShadowDom comme stratégie pour encapsuler le CSS du composant. Ainsi, des attributs uniques sont ajoutés sur chaque élément du fichier HTML afin de rendre le composant unique au niveau du style.

6.4.5 HOOK

Un composant à un cycle de vie décomposé en huit étapes, sur lesquels on peut agir en implémentant les méthodes disponibles sur ces hooks. Cependant tous ne sont pas utilisés pour des raisons de performances. Ainsi les développeurs devront, parmi tous les hooks disponibles, n'utiliser que deux :

- OnInit : interface qui permet d'implémenter la méthode ngOnInit() qui est appelée une fois le composant chargé. Cette méthode permet d'initialiser les variables, faire appel au serveur, souscrire aux Observable ;
- OnDestroy : Cette méthode est appelée à la destruction du composant et elle permet de « unsubscribe » aux Observables.

Unsubscribe aux Observables
Il est très fortement recommandé de réaliser cette étape pour éviter les effets de bords et se retrouver avec des erreurs incohérentes au niveau de données.

6.4.6 OBSERVABLES

Javascript est langage qui gère l'asynchrone de deux façons possibles, soit par les promesses soit par les Observables.

Nous recommandons d'utiliser les Observables notamment afin de profiter de la libraire RxJS.

Cette libraire va notamment permettre de faire de la programmation fonctionnelle à travers le chainage de fonction via les opérateurs RxJS. Ces opérateurs permettent d'agir sur un Observable afin de reformater la donnée et de la renvoyer en tant qu'Observable.

De plus, le store NgRX est basé sur les Observables.

6.4.7 IMMUTABILITE

Compte tenu des évolutions de l'EcmaScript et des apports du TypeScript, la mise en place d'une immutabilité dans un projet Angular ne devrait nécessiter aucune librairie ou utilitaire spécifique, hormis concernant la "mise à jour" des objets.

Ce dernier cas peut, dans le cas où le besoin soit récurrent, être couvert par un utilitaire minimaliste, tel que :

- https://github.com/kolodny/immutability-helper

6.4.8 GESTION DES CODES RETOUR HTTP

En informatique, le code HTTP (aussi appelé code d'état) permet de déterminer le résultat d'une requête ou d'indiquer une erreur au client.

- 200 : succès de la requête :
- 301 et 302 : redirection, respectivement permanente et temporaire ;
- 400 : la majorité des erreurs métiers détectées par le serveur
- 401 : utilisateur non authentifié ;
- 403 : accès refusé ;
- 404 : page non trouvée. Correspond également à une ressource non trouvée.
- 500 et 503 : erreur serveur : l'erreur 500 correspond au code erreur renvoyé par le serveur en cas d'erreur technique.
- 504 : le serveur n'a pas répondu.

Se référer à la documentation suivante : https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP

6.4.9 ACCES AU DOM – DATA BINDING

La manipulation du DOM ne doit jamais se faire avec du Javascript natif. Ne jamais utiliser les méthodes disponibles des web API tel que « querySelector », « appendChild » etc…

En effet avec Angular, c'est avec le concept de « data binding » que nous allons pouvoir manipuler le DOM et c'est le framework qui se chargera alors d'appeler les méthodes des web API à notre place et qui fera le travail nécessaire pour référencer et modifier le DOM. Pour cela nous allons utiliser une syntaxe propre au framework :

- L'utilisation des crochets « [ ] » qui permet de binder une propriété du DOM en lui passant du Javascript, comme ceci :

```
[src]="maVariable"
```

Ici, la propriété du DOM **src** est liée dynamiquement avec une variable dans le code typescript du composant.

6.4.10 GESTION DES EVENEMENTS – EVENT BINDING

La gestion des évènements ne doit jamais se faire avec les méthodes natives Javascript disponibles avec les web API tel que « addEventListener » etc…

Avec Angular la gestion des évènements se fait avec le concept de « Event binding » disponible avec le framework et qui se charge d'appeler les méthodes des web API.

Pour gérer les évènement avec Angular nous utilisons donc les parenthèses « ( ) » dans le template HTML, exemple :

```
<button (click)="onClickConnected()">Connexion</button>
```

Ici, l'évènement « onClick » sur le bouton est lié à une méthode définis dans le Typescript du composant afin de répondre et réaliser une tâche.

6.4.11 SERVICE

Dans Angular les services vont nous permettre de réaliser des requêtes vers un serveur afin de récupérer des données et les mettre à disposition sous forme d'Observables auxquels on pourra souscrire.

A la création du service, une annotation particulière est donnée par Angular : **@Injectable()**. Cette annotation est nécessaire (toute les classes doivent avoir une annotation) et dans le contexte de l'application cela permet au service de profiter de l'injection de dépendance. Le service va en effet avoir besoin d'au moins une dépendance qui est la classe **HttpClient**.

De plus, une fois provide dans un module, une instance du service est créée au sein du framework dans la « **factory** » afin qu'il puisse être mis à disposition pour tous les composants ayant besoin de ce service.

Ainsi, un service n'aura qu'une seule instance ce qui fait de lui un **singleton**.

6.5 CREATION DES COMPOSANTS

6.5.1 CREATION DE L'APPLICATION

L'application est créée par la commande suivante :

```
ng new my-app --routing
```

Une fois créée, il est nécessaire de se positionner dans le répertoire de celle-ci pour pouvoir générer les différents composants.

```
cd /chemin/my-app
```

6.5.2 CREATION D'UN MODULE FONCTIONNEL

Le module est créé de la façon suivante :

```
ng generate module features/domain-feature-fm --routing
```

Chaque module fonctionnel possède son propre module de routing interne lui permettant de gérer la navigation entre ses pages.

Attention : chacun des composants du module doit être exporté pour être visible de l'application.

6.5.3 CREATION D'UN DATA MODEL

Suivant le répertoire (propre à une features ou partagé), il est possible d'utiliser l'une des deux commandes.

Il est fort probable qu'une ressource du data model soit dans la partie partagée (2nde commande)

```
ng generate class features/donneur/models/my-datamodel-model --module donneur
ng generate class shared/donneur/models/my-datamodel-model
```

L'option « --module » permet d'éviter le rattachement à un sous-répertoire.

6.5.4 CREATION D'UNE ENUMERATION

Les énumération sont créées dans le répertoire shared/generic/enumerations

```
ng g enum shared/generic/enumerations/nomenum
```

6.5.5 CREATION D'UN SERVICE

```
ng generate service feature/domaine-feature-fm/dataservice/my-dataservice-ds --flat
```

6.5.6 CREATION D'UNE PAGE/CONTAINER

```
ng generate component features/domaine-feature/containers/my-page-container --module domaine-feature
```

6.5.7 CREATION D'UN FRAGMENT

```
ng g c shared/donneur/fragments/identite-fg --module shared
ng g c features/domaine-feature/fragments/identite-fg --module domaine-feature
```

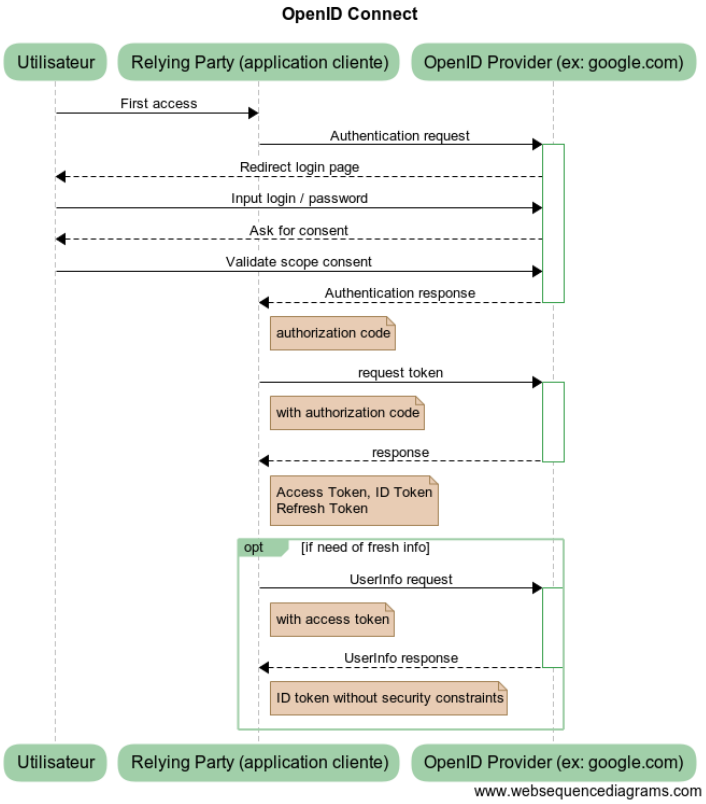
Répertoire des fragments

Les fragments sont créés dans le répertoire « ./fragments » du module fonctionnel.

6.6 GESTION DE L'AUTHENTIFICATION

6.6.1 DELEGATION DE LA GESTION DE L'AUTHENTIFICATION A LA LIBRAIRIE KEYCLOAK-JS

La librairie Keycloak-js est l'adapteur client fourni par Keycloak, permettant de sécuriser les applications et services en mettant en œuvre les protocoles OpenID Connect ou SAML ; elle permet de simplifier la gestion de tous les modes d'accès, comme l'Implicit flow ou l'authorization code flow, de manière transparente. Afin de se rendre compte des services qu'elle rend, voici le workflow de l'authorization code flow :



La librairie prend en compte tous les échanges indiqués ci-dessus. A noter que l'authentification s'effectue directement dans Keycloak.

Elle permet également de mettre en œuvre le SSO et les appels inter-applicatifs avec différentiation des jetons d'accès en fonction de l'application cliente.

La librairie est fournie sous deux déclinaisons, une version Angular et une version Javascript. La première est à utiliser intrinsèquement dans les applications Angular, la deuxième dans les applications JEE comme les applications monolithiques de l'Agence.

6.6.2 UTILISATION DE LA LIBRAIRIE KEYCLOAK-JS

En l'absence de la librairie Keycloak-angular (optionnelle), l'utilisation de Keycloak-js nécessite :

- La création d'un service angular : il a pour vocation à rendre accessible l'instance de Keycloak-js dans tous les composants ayant besoin de connaitre l'utilisateur connecté ou de récupérer les jetons nécessaires à l'appel des services du backend ;
- L'initialisation du service lors du démarrage de l'application dans le module « app.module.ts » de l'application ;

Dans le cas contraire, des éléments sont fournis par la librairie Keycloak-angular facilitant la configuration.

A cet effet, il faut utiliser paramétrer « onLoad » à « ***login-required*** » lors de l'initialisation de la librairie :

```
export const keycloakInit = async <T extends Configuration>(keycloak: KeycloakService, config: T): Promise<boolean> =>{
  return keycloak.init({
    config: {
      url: 'https://keycloak-host/auth',
      realm: 'client-keycloak-application',
      clientId: 'client-keycloak-application'
    },
    initOptions: {
      onLoad: 'login-required',
      checkLoginIframe: false,
      pkceMethod: 'S256',
      useNonce: true,
    }
```

6.6.3 });INJECTION DU TOKEN D'AUTHENTIFICATION DANS LES REQUETES HTTP

Afin de ne pas avoir à ajouter « manuellement » le token à chaque appel au BackEnd, Angular permet de mettre en place un mécanisme d'interception de toutes requêtes sortantes et de les manipuler pour y injecter toute information nécessaire. A cet effet, un service d'interception héritant de la classe HttpInterceptor doit être créé :

```
@Injectable({
  providedIn: 'root'
})
export class HttpInterceptorService implements HttpInterceptor {
  constructor(private authService: AuthentificationDsService, private router: Router) { }
  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const XApiKey = this.getApiKey();
    this.keycloakService.updateToken(30).then().catch(
catchError(error => {if (error instanceof HttpErrorResponse) {
                return throwError(error);
}}));
    if (this.keycloakService.getKeycloakInstance().token) {
      request = this.addApikeyAndTokenToRequest(
request,
```

```
this.keycloakService.getKeycloakInstance().token,
XApiKey
      );
    }
    return next.handle(request).pipe(
      catchError(error => {
        if (error instanceof HttpErrorResponse) {
          return throwError(error);
        }
      })
    );
  }}
```

Attention ce code est donné à titre indicatif et doit être adapté avec plus de contrôles avant et après injection.

- La méthode « this.keycloakService.updateToken(30) » de rafraichir le jeton d'accès s'il lui reste moins de 30 secondes avant expiration ou s'il est expiré et que le jeton de rafraichissement est toujours valide.
- La méthodes «this.keycloakService.getKeycloakInstance().token » permet de récupérer le jeton d'accès à injecter dans le header de la Request.



Interception des requêtes HTTP

Une fois le service d'interception créé, il est systématiquement appelé lors de l'emploi du HttpClient de Angular.

6.7 GESTION DES ACCES AUX RESSOURCES

6.7.1 OBJECTIF

L'accès aux ressources que constituent les écrans ou éléments de l'interface doit être soumises aux permissions octroyées par l'application à l'utilisateur connecté. Cet accès peut généralement se faire de trois manières distinctes :

- Par une action sur un élément de l'interface comme un lien html ou un bouton.
- Par programmation.
- Par modification du chemin dans la barre de navigation du navigateur.

De ce fait, le conditionnement peut être réalisés soit en masquant les éléments permettant l'accès, soit en affichant une erreur ou en redirigeant vers l'écran de connexion.

- Le masquage ne peut se faire que de manière spécifique.
- La redirection vers une page d'erreur ou l'écran de login peut se faire par le mécanisme des Routes Guard.



Attention
Ces mécanismes ne constituent en aucun cas des éléments de sécurités ; la gestion des accès aux ressources doit se faire au niveau des APIs HTTP

6.7.2 ROLES ET PERMISSIONS

Une fois l'utilisateur connecté, et le cas échéant, son rôle déterminé, l'application doit récupérer la matrice des permissions associées ; cette dernière permet de déterminer ce que l'utilisateur peut faire dans l'application.

6.7.3 MASQUAGE DES ELEMENTS DE L'INTERFACE

L'objectif est de ne pas afficher certains éléments comme des boutons, des liens, ou autre, si l'utilisateur ne dispose pas de cette permission.

A cet effet, et dans un but de centralisation, la gestion de ces derniers peut être déléguée à une directive Angular.

```
import { Directive, OnInit, OnDestroy } from '@angular/core';
import { Input, TemplateRef, ViewContainerRef } from '@angular/core';
import { AuthService } from '../auth/auth.service'

@Directive({
  selector: '[IfPermissionAllowed]'
})
export class CanAccessDirective implements OnInit, OnDestroy {
  @Input('IfPermissionAllowed') permission: string;
  constructor(
     private authService: AuthService,
     private templateRef: TemplateRef<any>,
     private viewContainer: ViewContainerRef) { }
     private subscription = new Subscription();

  ngOnInit() {
    this.subscription.add(
    this.authService.checkPermission(this.permission).subscribe(res => {
      if(res) {
        this.viewContainer.createEmbeddedView(this.templateRef);
      } else {
        this.viewContainer.clear;
      }
    })) ;
```

```
  }
  ngOnDestroy() {
    this.subscription.unsubscribe() ;
  }
}
```

La directive délègue la gestion des permissions au services d'authentification.

Et pour l'appliquer à un élément de l'interface comme un bouton par exemple :

```
<button *IfPermissionAllowed="'consult-dossier-donneur'"
  (click)="go('screen-one')"
>Screen One</button>
```

6.7.4 ROUTES GUARD

Les Routes Guard ont pour but de conditionner la navigation survenant en cliquant sur un lien, un bouton ou en modifiant l'url dans la barre d'adresse du navigateur, à un ensemble de permissions liées au rôle d'utilisateur connecté. Il s'agit avant tout de services Angular.

Exemple de protection de route :

```
const routes: Routes = [
  { path: 'screen-one',
    component: ScreenOneComponent,
    canActivate: [AuthGuard], data: { permission: 'consult-dossier-donneur' }
  },
  { path: 'screen-two',
    component : ScreenTwoComponent,
    canActivate : [AuthGuard], data : { permission : 'update-dossier-donneur' }
  },
  { path : 'error', component : ErrorComponent}
] ;
```

L'accès aux composants ScreenOne et ScreenTwo sont soumis à l'autorisation de la Route Guard et aux permissions « consult-dossier-donneur» et « update-dossier-donneur » .

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from
'@angular/router';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
import { AuthService } from '../auth/auth.service'
@Injectable({
  providedIn: 'root'
})
```

```
export class AuthGuard implements CanActivate {
  constructor(private loginDsService: LoginDsService,
     private router: Router,
     private keycloakService: KeycloakService
   ) {}

   canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
    69oolean | Observable<69oolean> | Promise<69oolean> {
     // tslint:disable-next-line:max-line-length
       if (this.keycloakService.getKeycloakInstance().authenticated &&
           this.keycloakService.getKeycloakInstance().token ) {
           return true;
       } else if (!this.keycloakService.getKeycloakInstance().authenticated) {
           removeAllItemsLocalstorage();
           this.router.navigate(['']);
           return false;
       } else {
           removeAllItemsLocalstorage();
           this.router.navigate(['']);
           return false;
    }
69oolean'' }
```

La Route Guard délègue la gestion des permissions à la librairie Keycloak-js.'' '' '' ''

Centralisation des règles d'accès

Les règles doivent être codées au cas par cas et prennent en entrée le contexte utilisateur et le contexte applicatif qui leur sont nécessaire.

Les règles seront normées.

6.8 LES ROUTES

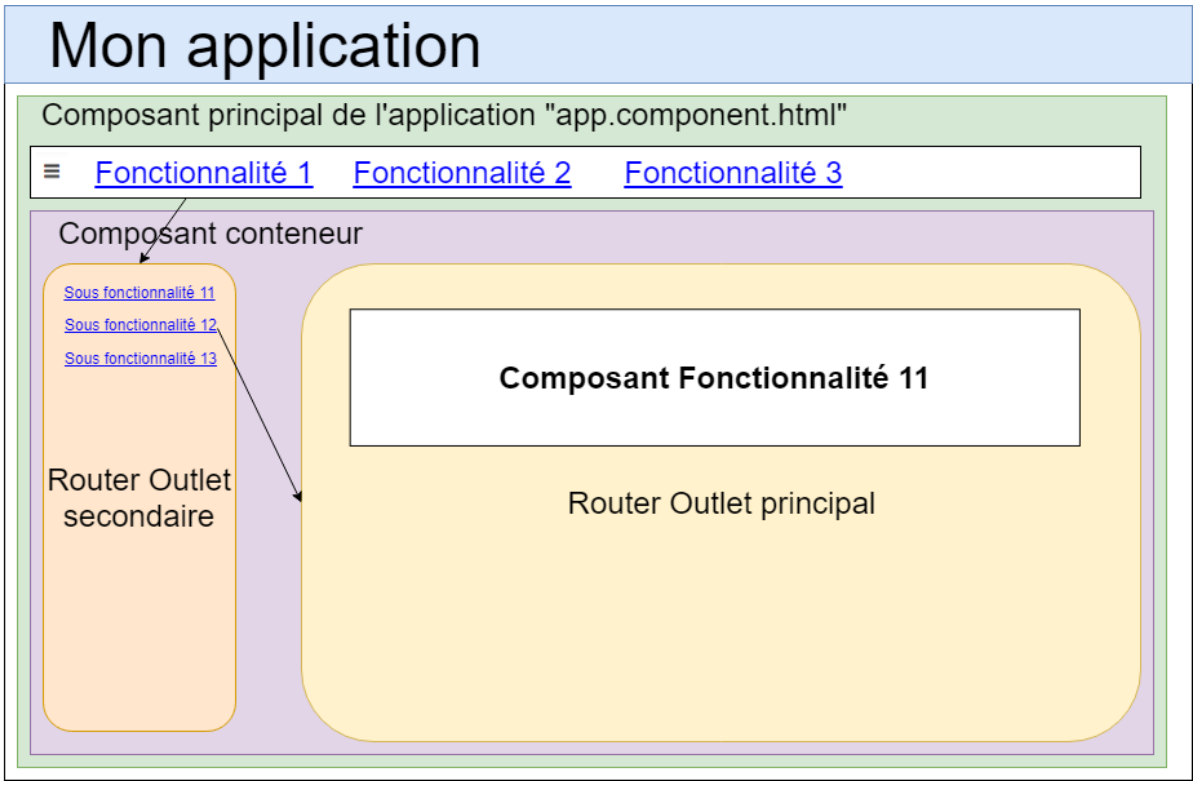
6.8.1 DESCRIPTION

Angular permet de définir la navigation entre écrans via le mécanisme de route mettant en relation une url et le composant à afficher.

L'affichage se fait alors dans un composant appelé le Router Outlet. Angular permet de définir un Router Outlet principal et des Router Outlets secondaires. Sans spécification particulière, les composants sont affichés dans le Routeur Outlet principal.

6.8.2 DEFINITION DES ROUTER OUTLET DE L'APPLICATION

L'application se décompose graphiquement de la manière suivante :



Le composant « app.component.html » de l'application contient un router. Ce dernier permet d'afficher le composant « Conteneur » de l'application possédant lui-même un router principal et un router secondaire.

- Le routeur principal affiche les composants associés aux sous fonctionnalités.
- Le routeur secondaire affiche le panneau latéral de navigation spécifique à chaque module et permettant l'accès aux sous fonctionnalités exposées.

Cette architecture permet d'afficher les différents composants dans le router principal et dans le router secondaire de manière très simple avec une url de la forme : « /fonctionnalité1/sousfonctionnalité11 »

Panneau latéral de navigation

Chaque fonctionnalité principale doit posséder son propre panneau de navigation latéral associé. Ce dernier est un composant Angular décrivant les liens d'accès aux composants associés aux sous fonctionnalités.

6.8.3 ROUTING FORROOT

Lors de la création d'une application Angular, le module « app-routing.module.ts » est généré automatiquement par Angular afin de mettre à disposition un système de routing pour le point de départ de l'application à savoir le component root : « app.component ».

Une fois les routes créées, le Router Outlet peut alors être utiliser dans le template du component root. Ainsi, le composant principal de l'application « app.component.html » ne contient qu'un seul routeur :

```
<layout
  [titleHeader]="'SYRENAD – DONNEUR'"
  [infoCurrentUser]="infoCurrentUser"
  [menuItems]="itemsMenu"
  [indexDefaultMenuActive]="0"
  (selectedMenu)="getSelectedMenu($event)" (logout)="onLogout($event)">
  <router-outlet></router-outlet>
</layout>
```

Ce **Router Outlet** présent dans le « app.component.html » est le point d'entrée de l'application qui va permettre à tous les autres composants de se charger en fonction de feature à laquelle ils appartiennent, feature qui possède son propre système de routing.

Ce Router-outlet du « app.component » est dit « **forRoot** ». En effet un système de routing est créé (un objet **Routes**) dans le fichier « app-routing.module.ts » et est importé en tant que « forRoot », forRoot désignant le app.component qui est le point d'entrée d'une application Angular. Le routing :

```
const routes: Routes = [
  {path: '', redirectTo: 'auth/connexion', pathMatch: 'full'},
  {path: 'auth', redirectTo: 'auth/connexion', pathMatch: 'full'},
  {path: 'main', redirectTo: 'main/tableau-de-bord', pathMatch: 'full'},
  {path: 'main/donneur-att', redirectTo: 'main/donneur-att/liste-donneur-att', pathMatch: 'full'}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {
}
```

6.8.4 ROUTING FORCHILD

Un deuxième niveau de routing est mis en place dans l'application, il concerne cette fois les fonctionnalités : chaque fonctionnalité aura son propre système de routing.

La commande suivante, permet la création du routing en même temps que la création de la nouvelle feature. Ce routing sera qualifié de « **ForChild** » :

```
ng generate module feature/domain-feature-fm –routing
```

Définition des routes

Chaque module fonctionnel possède son module de gestion des routes.

Exemple du module contenant les routes :

```
const routes: Routes = [
  {
    path: '', children: [
      {path: 'liste-donneur-att', component: ListeDonneurAttComponent}
    ]
  },
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class DonneurAttRoutingModule {
}
```

Ici, l'objet Routes est cette fois importé en tant que « ForChild » et est donc disponible uniquement pour la fonctionnalité.

Cette gestion de routing par fonctionnalité va permettre de charger les composants à la demande comme il sera présenté plus tard dans la partie « Lazy Loading ».

Cette route définit les routes secondaires d'accès aux sous fonctionnalités du module 1.

Nommage des routes

Attention, afin d'éviter des priorisations de routes, il est important d'avoir des routes uniques et ne constituant pas des sous chaines d'une autre route existante.

Accès à des composants secondaires

Si une sous fonctionnalité permet de visualiser des composants secondaires, comme par exemple accéder à un détail à partir d'une liste, les routes doivent en tenir compte.

6.8.5 UTILISATION DES ROUTES

Les routes peuvent être activées de plusieurs manières :

- Soit via l'utilisation de liens :

```
<a [routerLink]= »[ 'fonct1fm'] » routerLinkActive= »active »>Fonctionnalité 1</a>
```

- Soit via un bouton :

```
<button [routerLink]= »[ 'fonct1fm'] »>Fonctionnalité 1</button>
```

- Soit via la programmation :

```
this.router.navigate(['/sousfonct1']) ;
```

6.8.6 PASSAGE DE PARAMETRES

Il est possible de passer des paramètres via les routes et de les récupérer dans le composant appelé, comme par exemple l'identifiant de l'élément d'une liste afin d'en afficher le détail.

Cela se traduit dans le fichier de routing par :

```
{ path: 'detail-pn/:id', component: DetailPnComponent},
```

Puis lors de l'appel de la route :

- Par routerLink

```
<button [routerLink]="['sousfonct1','1'] button>
```

- Par programmation :

```
this.router.navigate(['/sousfonct1',  id]);
```

Et enfin, la récupération de l'identifiant dans le composant « DetailPnComponent » :

```
 ngOnInit() {
   this.subscription.add(
    this.route.paramMap.subscribe( params => {
      if(params.has('id')) {
        this.getDetail(+params.get('id'));
```

```
      }
    }));;
  }
```

6.8.7 LAZY LOADING

Le lazy loading va permettre de charger à la demande des modules en fonction des actions utilisateurs (click vers une route).

Il y en effet deux possibilités lorsqu'on visite les routes d'une application Angular :

- Charger tous les modules au lancement de l'application
- Charger certains modules nécessaires à l'application et les autres à la demande

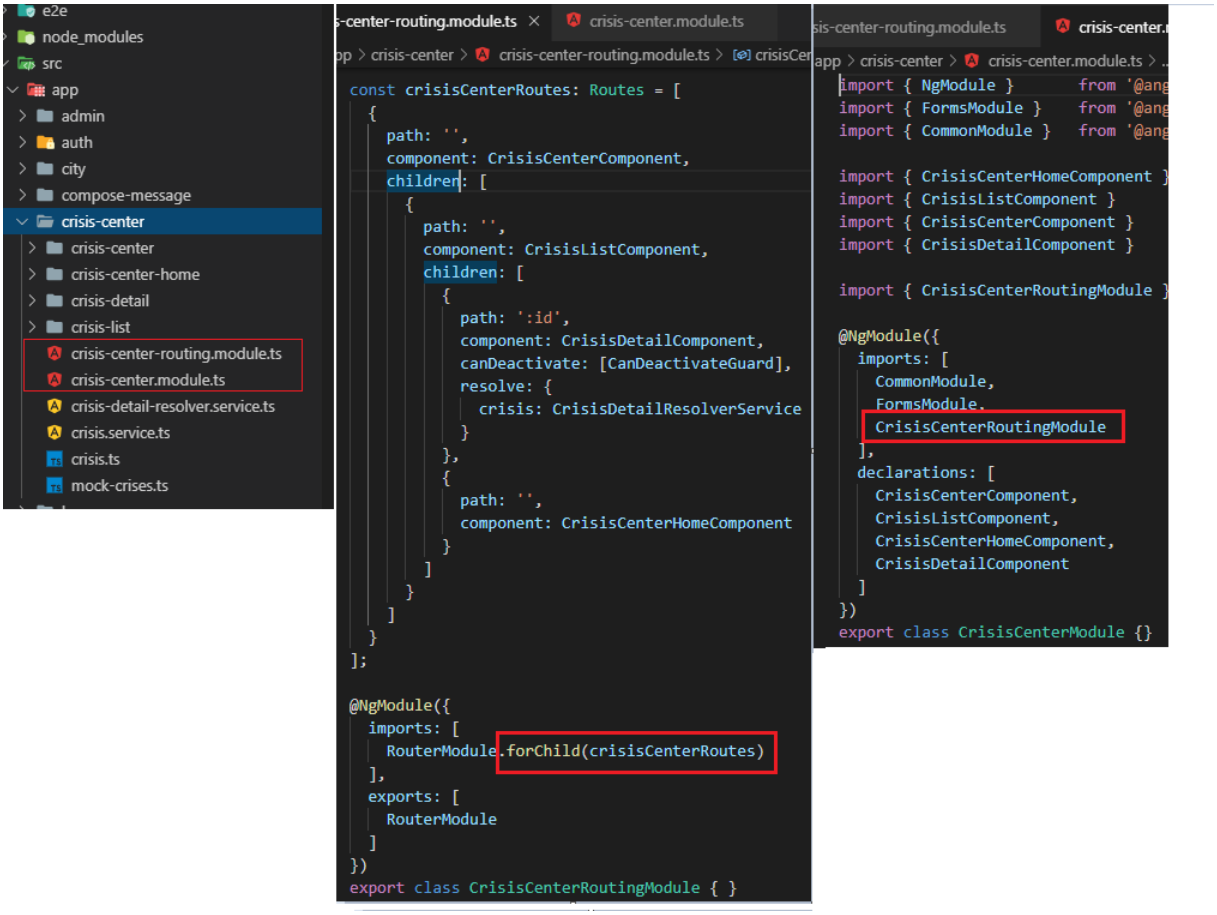
Pourquoi le lazy loading ?

Cette bonne pratique va permettre d'optimiser l'application et ainsi avoir des meilleures performances avec un chargement plus rapide.

Comment appliquer le lazy loading ?

Pour mettre en place le lazy loading on commence par créer pour chaque feature son propre routing ainsi que son propre module. Ainsi, chaque feature sera responsable de ses routes. Un exemple de la documentation officielle de Angular. Dans l'exemple ci-dessous nous allons gérer la feature « crisis-center » :

1- Créer un fichier module pour la feature et un fichier routing
2- Dans le fichier routing de la feature définir ses routes.
3- Importer le module de routing dans le module principal de la fetaure



Ensuite, ne pas importer le module du feature dans le app.module.ts comme on le ferait normalement pour un feature charger dès le lancement de l'application mais utiliser la propriété « loadChildren » dans le fichier app-routing pour la route de la feature « CrisiCenter » comme ceci avec le chemin vers le module de la feature :

app-routing.module.ts

src > app > app-routing.module.ts > AppRoutingModule

```
1   import { NgModule }             from '@angular/core';
2   import { RouterModule, Routes } from '@angular/router';
3
4   import { ComposeMessageComponent }  from './compose-message/compose-message.com
5   import { PageNotFoundComponent }    from './page-not-found/page-not-found.compo
6
7   import { AuthGuard }                          from './auth/auth.guard';
8   import { SelectivePreloadingStrategyService } from './selective-preloading-stra
9
10  const appRoutes: Routes = [
11  >  {…
15     },
16  >  {…
20     },
21     {
22       path: 'crisis-center',
23       loadChildren: './crisis-center/crisis-center.module#CrisisCenterModule',
24       data: { preload: true }
25     },
26     // {
27     //   path: 'cities',
28     //   loadChildren: () => import('./crisis-center/crisis-center.module').then(
29     //   data: { preload: true }
30     // },
31     { path: '',   redirectTo: '/superheroes', pathMatch: 'full' },
32     { path: '**', component: PageNotFoundComponent }
33  ];
34
35  @NgModule({
36    imports: [
37      RouterModule.forRoot(
38        appRoutes,
39        {
40          enableTracing: false, // <-- debugging purposes only
41          preloadingStrategy: SelectivePreloadingStrategyService,
42        }
43      )
44    ],
```

Ainsi, ce n'est que lors de la visite de route « crisis-center » que le module sera chargé.

6.9 LES FORMULAIRES

Angular apporte une gestion des formulaires permettant de garder la trace des saisies de l'utilisateur, de garder l'état de validité des informations saisies et enfin d'afficher les messages d'erreur ; à cet effet deux types de gestions sont possibles :

- Les formulaires de type « template driven forms» : leur définition se fait dans le template du composant.
- Les formulaires de type « reactive forms » : leur définition se fait dans le contrôleur du composant.

Ces deux modes permettent le binding bi directionnel des données.

Choix d'un type de gestion

Il est recommandé d'utiliser une gestion de type « Reactive Form » afin de permettre d'effectuer des tests unitaires ce que n'autorise pas les « template driven ».

6.10 GESTION DES ERREURS

6.10.1 OBJECTIF

L'objectif est de centraliser la gestion des erreurs de contrôle de forme (champ obligatoire, d'une taille minimale ou maximale, etc.) afin ne pas à devoir les gérer manuellement à chaque champ de formulaire.

A cet effet, deux composants sont chargés de cette centralisation :

- Un composant chargé d'afficher les messages à l'écran.
- Une directive chargée de contrôler chaque champ de formulaire.

6.10.2 IMPLEMENTATION

- Le composant d'affichage des erreurs

```
import { Component, OnInit, Input } from '@angular/core';
@Component({
  selector: 'app-error',
  template: `<p-message severity="error" text="{{_text}}"></p-message>`
})
export class ErrorComponent implements OnInit {
  _text: string;
  constructor() { }
  @Input() set text(value: string) {
    this._text = value;
  }
  ngOnInit() {
  }
}
```

Ce composant encapsule l'objet « p-message » de PrimeNG.

- La directive de gestion et de centralisation des erreurs :

```
import { Directive,
ElementRef,
ViewContainerRef,
OnInit,
```

```
Self,
ComponentRef,
ComponentFactoryResolver } from '@angular/core';
import { NgControl } from '@angular/forms';
import { distinctUntilChanged } from 'rxjs/operators';
import { ErrorComponent } from './error.component';

@Directive({
  selector: '[formControl],[formControlName]',
  host: {
    '(blur)': 'onBlur($event)'
  }
})
export class ErrorDirective implements OnInit {

  ref: ComponentRef<ErrorComponent>;
  constructor(
private componentFactoryResolver: ComponentFactoryResolver,
private vc: ViewContainerRef,
private elem: ElementRef,
@Self() private control: NgControl ) { }
  ngOnInit() { }
  onBlur($event: any) {
    this.checkErrors();
  }
  displayErrorComponent(value: string) {
    if(!this.ref) {
      const factory = this.componentFactoryResolver.resolveComponentFactory(ErrorComponent);
      this.ref = this.vc.createComponent(factory);
    }
    this.ref.instance.text = value;
  }
  hideErrorComponent() {
    if(this.ref) {
      this.ref.destroy();
      this.ref = null;
    }
```

```
  }
  checkErrors() {
    if(this.control.hasError('required')) {
      this.displayErrorComponent('Le champ est obligatoire !');
    } else {
      this.hideErrorComponent();
    }
  }
}
```

- Attention, seule la propriété « Required » est ici prise en compte dans l'exemple ci-dessus.
- La directive s'applique directement au [formControl] et [formControlName] sans avoir besoin de la déclarer au niveau des champs :

```
<textarea [rows]="5" [cols]="30" pInputTextarea autoResize="autoResize"
formControlName="description"></textarea>
```

Définition d'un champ ayant l'attribut « Required » :

```
this.projetForm = this.fb.group({
 'nom': new FormControl(projet.nom, { validators: Validators.required, updateOn: 'blur' })});
```

A noter la propriété « updateOn: 'blur' » qui permet de ne déclencher le contrôle que lorsque l'utilisateur sort du champ.

Le résultat si le champ est laissé vide est le suivant :



6.11 LES TESTS UNITAIRES

A préciser

7 Gestion de la configuration

7.1 FRONTEND

7.1.1 GESTIONNAIRE DES PAQUETS

Le rôle du gestionnaire de paquets est de gérer l'installation des paquets nécessaires au développement d'une application Angular.

Les deux majeurs gestionnaires de paquets dans l'écosystème Angular sont NPM et Yarn. Ils peuvent être utilisés de concert.

Ils téléchargent les librairies nécessaires et mettent à jour le fichier de dépendance de l'application « package.json ».

- NPM permet de gérer des paquets locaux à une application ou globaux.
- Yarn est plus rapide à télécharger les librairies et est en mesure de mettre en cache ces dernières.

7.1.2 GESTION DE LA CONFIGURATION DE L'APPLICATION

La configuration de l'application englobe le paramétrage des appels des services Web, des appels au serveur d'autorisation et de la valeur de la clé de l'API d'authentification. Exemple :

```
{
  production: false,
  webresturl: 'http://calypso:10080/apiman-gateway/abm/green/1.0/',
  securitytoken: 'http://calypso.natsystem.fr:10080/auth/realms/green/protocol/openid-connect/token',
  apikey: '9634f559-528a-4c0d-bf65-d6dea088e012'
}
```

Ces paramètres dépendent de l'environnement de déploiement et doivent donc pouvoir être modifiés facilement.



Fichier de configuration Angular
Par défaut, Angular propose de gérer la configuration via deux fichiers générés à la création du projet : environment.ts et environment.prod.ts. Malheureusement, une fois le build du projet effectué, il n'est plus possible de changer leur contenu. Ils ont donc à éviter dans le cadre d'une application nécessitant un paramétrage adapté à l'environnement de déploiement.

Pour rendre la configuration dynamique, cette dernière sera placée dans un fichier json rendu disponible par injection dans les providers de l'application.

Ce fichier « config.json » sera créé dans le répertoire « assets » de l'application.



Répertoire « assets »

Ce répertoire permet de rendre disponible des ressources comme des images ou autre et est build tel quel par Angular, c'est-à-dire que les fichiers ne sont pas transformés, entre autres, ils ne sont pas minifiés.

La lecture du fichier se fait par l'utilisation de l'API fetch de javascript, ce chargement est effectué au chargement de l'application (voir chapitre 8)

Voici un extrait de la factory en charge de l'initialisation de l'application, qui va notamment lire le fichier de configuration au format JSON :

```
const configUrl = './assets/configurations/config.json';

return fetch(configUrl)
  .then(response => response.json())
  .then(configJson => configLoad(httpclient, configurationService, configJson, ConfigurationDonneur))
  …
```

La fonction « configLoad » s'occupe d'affecter les valeurs du fichier de configuration au service applicatif « ConfigurationService » afin de pouvoir accéder aux éléments de configuration n'importe où dans l'application :

```
import {HttpClient} from "@angular/common/http";
import {ConfigurationService} from "@syrenad-front-monorepo/shared/utilisateur";

export const configLoad = async <T>(httpClient: HttpClient, configurationService:
ConfigurationService, configJson: any, c: { new (): T }): Promise<T> => {
 if (![null, undefined, ''].includes(configJson)) {
  configurationService.config = Object.assign(new c(), configJson);
  return configurationService.config as unknown as T;
 }
 throw new Error("Le fichier de configuation n'a pas été chargé");
};
```

Enfin le service de configuration doit être injecté là où on en a besoin comme les services d'appel au Backend :

```
constructor(private http: HttpClient, private _configService: ConfigurationService){) {
        this._uri = this. configService.config.uri;
}
  }
```

8 Gestion de l'initialisation de l'application

Il n'est possible d'utiliser qu'un seul APP_INITIALIZER dans une application Angular. Il faut donc que l'initialisation de l'authentification Keycloak soit est effectué dans la même factory que celle du chargement de l'application.

On a donc la factory suivante :

```
function initializeAppFactory(keycloakService: KeycloakService,
                    httpclient: HttpClient,
                    configurationService: ConfigurationService): () => Promise<boolean> {

  return (): Promise<boolean> => {
    const configUrl = './assets/configurations/config.json';

    return fetch(configUrl)
      .then(response => response.json())
      .then(configJson => configLoad(httpclient, configurationService, configJson,
ConfigurationDonneur))
      .then(config => keycloakInit(keycloakService, config))
      .catch(reason => {
        console.log('initializeApp failed :', reason);
        keycloakService.logout().then();
        throw new Error(`initializeApp failed : ${reason}`);
      });
  };

}
```

Et dans les providers, la nouvelle factory « initializeAppFactory» remplace la factory « APP_INITIALIZER » :

```
providers: [
  {
    provide: APP_INITIALIZER,
    useFactory: initializeAppFactory,
    deps: [KeycloakService, HttpClient, ConfigurationService],
    multi: true
  }
]
```