**Spécification technique – Guide de développement**

**support d’une solution open source d’ENT pour les EPLE**

**de la région Île-de-France**

**Spécification Technique   
Guide de développement**

**Auteur** :

**Version** : 1.0



**Gestion des changements de version**

*Ce tableau gère les modifications apportées au document au-delà de sa version initiale. Les petites modifications de type erreurs de frappe ou changements de syntaxe ne font pas l’objet d’un suivi. Toute nouvelle version du document ne conserve pas systématiquement les changements apportés lors de la version précédente.*

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Auteur** | **Objet de la mise à jour** |
| 1.0 | 27/06/2011 | MMAU | Initialisation du document |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Sommaire

[Introduction 5](#_Toc297827460)

[1. Règles générales 5](#_Toc297827461)

[1.1. Nomenclature 5](#_Toc297827462)

[1.2. Commentaires 6](#_Toc297827463)

[1.3. Normes externes 7](#_Toc297827464)

[1.4. Préconisations, codes à risques 7](#_Toc297827465)

[1.5. Constantes 8](#_Toc297827466)

[1.6. Utilisation du socle 8](#_Toc297827467)

[1.6.1. Logs 8](#_Toc297827468)

[1.6.2. Configuration du projet 8](#_Toc297827469)

[1.6.3. Autres utilitaires du socle à utiliser selon le besoin 8](#_Toc297827470)

[1.6.4. Librairies externes 9](#_Toc297827471)

[2. Organisation générale du back-office 9](#_Toc297827472)

[2.1. Arborescence générale 9](#_Toc297827473)

[2.2. Développement en couches 9](#_Toc297827474)

[3. Développement de la couche persistance 10](#_Toc297827475)

[3.1. Présentation 10](#_Toc297827476)

[3.2. Implémentation des DAO 11](#_Toc297827477)

[3.2.1. Classes de référence 11](#_Toc297827478)

[3.2.2. Héritage et nomenclatures 11](#_Toc297827479)

[3.2.3. Exceptions 12](#_Toc297827480)

[3.2.4. Utilisation d’Ibatis 12](#_Toc297827481)

[3.2.5. Définition des requêtes dans les SqlMap 13](#_Toc297827482)

[3.3. Configuration de la datasource 16](#_Toc297827483)

[3.4. Utilisation de POJO 16](#_Toc297827484)

[4. Développement de la couche métier 17](#_Toc297827485)

[4.1. Présentation 17](#_Toc297827486)

[4.2. Implémentation des classes métiers 17](#_Toc297827487)

[4.2.1. Classes de référence 17](#_Toc297827488)

[4.2.2. Héritage et nomenclatures 18](#_Toc297827489)

[4.2.3. Appel des autres couches 19](#_Toc297827490)

[4.2.4. Gestion des contrôles fonctionnels 19](#_Toc297827491)

[4.2.5. Appel de Web Services 20](#_Toc297827492)

[4.3. Les objets métiers 21](#_Toc297827493)

[4.3.1. DTO 21](#_Toc297827494)

[4.3.2. Adapter 22](#_Toc297827495)

[4.4. Gestion du cache 24](#_Toc297827496)

[5. Transactions 24](#_Toc297827497)

[6. Constantes et utilitaires 24](#_Toc297827498)

[Annexe 24](#_Toc297827499)

# Introduction

Le but de ce document est de rappeler certaines normes de développement Java ainsi que celles qui sont spécifiques au projet ENT. Ce document permet d’énoncer les règles à suivre mais il fait également référence aux différentes documentations du socle, afin de donner la marche à suivre pour l’intégration des différents éléments et outils à disposition.

Ce document décrit la partie back-office d’une application, c'est-à-dire le développement des classes métiers et DAO d’une api, d’un batch ou d’une appli web. La partie frontale est décrite dans une autre documentation spécifique au développement d’un module web Lilie.

Les règles générales communes à toutes les couches de toutes les applications sont également dans ce document.

# Règles générales

## Nomenclature

Voici les différentes règles de nommage à suivre concernant les noms des méthodes et des variables :

* Le code source du projet ENT est en français. En dehors des mots utilisés par le langage, et à part quelques exceptions pour des questions de facilités relatives à Java (le mot « *get* » par exemple est assez classique et accepté), il est bien d’éviter des mélanges de type « franglais », afin de rester homogène.

Quelques exemples :

* *getPersonneParNom()* plutôt que *getPersonneByNom()*
* *ajoutePersonne()* ou *creePersonne()* plutôt que *addPersonne()*
* Utiliser des noms ayant un sens fonctionnel utile, pas de numéros ou de lettres seules. On doit savoir à quoi sert l’objet rien qu’à son nom.
* Eviter les abréviations et le tronquage des mots à part quelques exceptions évidentes (« *supp* », « *maj* »…).
* Pour faire simple et homogène, pour les variables (locales, attributs, paramètres…), utiliser le nom du type de l’objet, qui devrait déjà être lui-même parlant ; ou pour les types primitifs, le mettre en suffixe.

Par exemple :

* *PersonneDto personneDto = new PersonneDto() ;*
* *ForumBusiness forumBusiness = new ForumBusinessImpl() ;*
* Ou bien si l’on effectue une modification et que l’on a besoin de deux instances, ajouter du sens fonctionnel en plus du type, par exemple : *anciennePersonneDto* et *nouvellePersonneDto*.
* *long personneIdLg = 2 ;*

(long : Lg ; boolean : Bln ; String : Str ; Date : Date ; Integer : Int )

* Pour les listes, on utilise le suffixe *List* (pour les map, le suffixe *Map*), et on reprend également le type de l’objet contenu dans la liste:

*List < PersonneDto > archivesPersonneDtoList = new ArrayList < PersonneDto >()*

* Réutiliser ces noms de type également dans les méthodes afin que l’on visualise rapidement le type de retour et le paramètre principal, par exemple :
* *public Long recherchePersonneIdLgParLoginStr(String loginStr, boolean archiveBln)*
* *public void ajoutePersonneDtoList(List < PersonneDto> personneDtoList)*
* Rester homogène entre les différentes méthodes d’une même classe ou d’un même projet : garder le même type de syntaxe, toujours un infinitif ou toujours un nom, le même ordre des mots…etc :

Si une méthode s’appelle *ajoutePersonneDto()*, ne pas en appeler une autre *rechercherPersonneDto()* ou *personneDtoRecherche()* mais plutôt

*recherchePersonneDto()*.

* Les noms des constantes doivent être en majuscules, chaque mot étant séparé par un « \_ ».
* Toutes les autres variables doivent commencer par une minuscule, avec une majuscule à chaque nouveau mot, sans « \_ » ou « - ».

## Commentaires

Les commentaires sont indispensables à la clarté du code, à son utilisation par une tierce personne et à sa maintenabilité. Il existe deux types de commentaires :

* la Javadoc qui est obligatoire et doit être explicite (dont quadrigramme de l’auteur dans le commentaire de classe)
* les commentaires de blocs de code qui aident à comprendre ce que fait un traitement ou à diviser le code en sous-parties.

Les commentaires doivent servir à comprendre le code sans le lire, un commentaire Javadoc copié-collé du nom de l’attribut ne sert pas à grand-chose, il faut expliquer.

De plus, un commentaire de License est obligatoire en en-tête de chaque fichier Java :

/\*

\* Copyright © LOGICA and the Conseil Régional d'Île-de-France, 2009

\*

\* This file is part of L'Interface Libre et Interactive de l'Enseignement (Lilie).

\*

\* Lilie is free software. You can redistribute it and/or modify since

\* you respect the terms of either (at least one of the both license) :

\* - under the terms of the GNU Affero General Public License as

\* published by the Free Software Foundation, either version 3 of the

\* License, or (at your option) any later version.

\* - the CeCILL-C as published by CeCILL-C; either version 1 of the

\* License, or any later version

\*

\* There are special exceptions to the terms and conditions of the

\* licenses as they are applied to this software. View the full text of

\* the exception in file LICENSE.txt in the directory of this software

\* distribution.

\*

\* Lilie is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* Licenses for more details.

\*

\* You should have received a copy of the GNU General Public License

\* and the CeCILL-C along with Lilie. If not, see :

\* <http://www.gnu.org/licenses/> and

\* <http://www.cecill.info/licences.fr.html>.

\*/

## Normes externes

Les normes de code Java, JSP et JS sont à respecter afin de garder un code propre et maintenable. Les documentations concernant ces normes externes sont mises à disposition avec les documentations techniques ENT.

Aucun warning Checkstyle ne doit rester lors du commit des sources sur le serveur. Le fichier de configuration Checkstyle par défaut a été adapté pour les besoins de l’ENT. Il est disponible dans le projet « tools ».

## Préconisations, codes à risques

* En plus des normes définies, il faut veiller à garder un code clair et lisible, bien indenté
* Faire des sous-fonctions pour factoriser le code ou pour le rendre plus compréhensible
* Faire attention aux chargements en mémoire (découper des méthodes trop longues pour éviter que les espaces mémoires ne soient vidés qu’à la fin)
* Utiliser *StringBuilder* pour les concaténations de chaînes de caractères
* Ne pas mettre d’appel de méthode dans un *for()*
* Penser aux tests de nullité pour éviter les *NullPointerException*
* Pour les *Enum*, utiliser des == et des *switch,* et non des *equals* et des *if*
* Eviter les tests inutiles du type :

*if (booleen) return true; else return false*. Retourner directement le booléen.

## Constantes

Les constantes et libellés ne doivent jamais être en dur dans le code. Ils doivent être externalisés, soit dans des constantes Java, soit dans des fichiers de ressources externalisables et internationalisables. Il en va de même pour les clés des propriétés dans ces fichiers de ressources : les clés doivent être mises dans des constantes Java afin de ne pas avoir à les modifier dans chaque partie du code si elles changent.

## Utilisation du socle

De manière générale, il est demandé d’utiliser au maximum les utilitaires proposés par le socle ENT. Cela permet de garder une homogénéité et de permettre la réutilisation de traitements communs. Les fonctionnalités proposées par le socle (notamment les apis et les projets core) sont décrites dans les documentations SPECHTECH-SocleApplicatif et SPECHTECH-frameworks-core. Le mieux est de les lire avant tout développement afin d’être conscient des différents outils proposés.

Certains sont repris dans les paragraphes suivants.

### Logs

Pour tracer des informations avec les différentes niveux de trace, il faut utiliser le logger de l’ENT *org.lilie.socle.core.utils.log.Logger*, en n’oubliant pas de préciser le bon nom de classe à chaque fois.

*private static final Logger LOGGER = Logger.getLogger(EleveDaoImpl.class);*

### Configuration du projet

Les fichiers de trace et de paramétrage du projet sont gérés de manière centralisés. Le fonctionnement à mettre en place dans chaque module ou batch est décrit dans la documentation du fmk-core-ent (et dans la documentation général du socle pour ce qui est commun à toutes les apis).

### Autres utilitaires du socle à utiliser selon le besoin

* Apis d’accès aux différentes parties du socle
* Gestion de l’utilisateur connecté
* Flux d’échanges de données entre modules
* Gestion des fichiers à uploader sur le serveur
* Api FTP
* Cryptage
* Email (SMS à venir)

### Librairies externes

Afin de garder une cohérence technique entre les projets mais également de simplifier la maintenance et la compatibilité des uns avec les autres, il faut au maximum mutualiser les librairies externes utilisées. Pour un besoin particulier, il vaut toujours mieux commencer par utiliser une librairie qui est déjà présente dans les dépendances ENT. Si une nouvelle librairie ou une autre version est nécessaire, il faut que cela se justifie et étudier également la possibilité de faire évoluer le reste de l’ENT vers cette librairie, plutôt que d’en garder plusieurs.

# Organisation générale du back-office

## Arborescence générale

L’arborescence du projet est basée sur les standards Maven. L’ENT utilise en effet Maven2 pour le packaging et le déploiement, il convient donc de respecter au maximum les standards de cet outil afin que les builds soient facilement automatisables :

* *src/main/java* : les sources java
* *src/main/resources* : les ressources diverses et configurations utlisées par les classes java
* *src/main/assembly* : pour les descripteurs des attaches à joindre au build
* *src/test/java* et *src/test/resources* : pour les classes de test du projet
* *pom.xml* à la racine du projet

## Développement en couches

Les projets ENT suivent les règles classiques de développement en couches. Chaque couche doit être au maximum indépendante des autres afin d’être plus facilement maintenable et interchangeable. Dans la partie back-office des ENT, nous distinguons deux couches principales : la couche métier et la couche persistance. Les règles de développement associées à chacune sont décrites dans les chapitres suivants.

Le découpage au sein du package de chaque couche est un découpage plus fonctionnel selon les données et/ou fonctionnalités gérées par le projet.



# Développement de la couche persistance

## Présentation

Les DAO permettent d’interagir avec une source de données. Ils ne doivent pas contenir de traitement métier. Ils doivent se limiter le plus possible à un traitement unitaire de récupération d’un type de données ou de modification de cette donnée en base. Cependant, pour des raisons de performances avec le SQL, on se retrouve de temps en temps contraints d’avoir un peu de métier dans la requête.

Afin de pouvoir évoluer vers une autre solution d’accès, pour chaque DAO on utilise des interfaces permettant de ne montrer que la signature des méthodes disponibles en cachant l’implémentation réelle à la couche supérieure.

Le lien entre la couche supérieure et l’implémentation est fait par l’utilisation du concept d’IoC de Spring : la couche supérieure ne crée pas directement l’objet représentant la couche inférieure qu’elle utilise, c’est Spring qui s’en charge par injection. Cela permet par exemple de changer facilement l’implémentation utilisée, la couche supérieure ne se souciant que de ce que lui fournit l’interface.

La plupart des DAO de l’ENT seront des implémentations Ibatis pour des accès à la base de données. Cependant, d’autres types de besoin peuvent être rencontrés au cours du développement. De manière générale dans l’ENT, un seul projet contient les DAO concernant une base de données particulière. Ainsi pour chaque base commune, on crée une api qui centralise le tout; et chaque module a sa propre base pour ses données spécifiques.

## Implémentation des DAO

### Classes de référence

Le package pour la couche DAO est le suivant :

*org.lilie.socle.api.<projet>.dao* pour une api

*org.lilie.services.<module>.dao* pour un module

*org.lilie.socle.bat.<projet>.dao* pour un batch

Un sous-package « reference » permet de mettre les différentes classes de référence du projet, c'est-à-dire les classes mères de la couche DAO, qui servent de « modèles ».

La classe de référence *IbatisDaoImpl* de chaque projet hérite de celle de *fmk-core-ent* qui permet de donner accès aux différentes fonctionnalités d’Ibatis. Elle permet de centraliser les traitements communs éventuels des DAO du projet. Un bean Spring est défini afin de centraliser la déclaration de l’attribut Ibatis (exemple pour l’api-annuaire) :

*<bean id="annuaireParentDao" abstract="true">*

*<property name="sqlMapClient" ref="annuaireSqlMap" />*

*</bean>*

D’autres classes de référence pour d’autres types de source de données pourraient éventuellement être ajoutées.

### Héritage et nomenclatures

Un DAO représente en général une fonctionnalité ou un type de données. Il est possible également d’ajouter un découpage en plusieurs sous-packages, afin d’organiser en différents domaines si les DAO sont nombreux.

Comme précisé au-dessus, le DAO est composé de deux fichiers, l’interface et l’implémentation. L’interface doit se nommer *<fonctionnalite>Dao.java*  et être dans la package racine « *dao* ». L’implémentation doit se nommer *<fonctionnalite>DaoImpl.java*, ou *<fonctionnalite>DaoIbatisImpl.java* par exemple si l’on veut préciser le type d’implémentation. Les implémentations se mettent dans un sous-package « *impl* ».

L’interface ne contiendra que la signature des méthodes déclarées *public* devant être disponibles pour la couche métier. Les méthodes d’implémentation n’étant utilisées qu’en local doivent être déclarées en *private* dans la classe d’implémentation.

La classe d’implémentation doit faire un *implements* explicite de son interface.

Il faut déclarer chaque DAO dans la configuration Spring afin qu’un bean correspondant soit disponible pour les couches supérieures qui veulent l’utiliser. Dans le fichier *persistance-applicationContext.xml*, on ajoute l’élément suivant (exemple pour l’api-annuaire):

*<bean id="eleveDao" class="org.lilie.socle.api.annuaire.dao.impl.EleveDaoImpl"/>*

C’est cette ligne qui fait le lien entre l’interface appelée par le métier et l’implémentation réellement utilisée. Le nom du bean doit reprendre celui de l’interface et non celui de l’implémentation afin de garder les couches étanches.

Chaque implémentation doit également étendre la classe mère *IbatisDaoImpl* du projet afin d’hériter des fonctionnalités d’accès à la base de données. Il faut donc ajouter la dépendance également dans la définition du bean Spring :

*<bean id="eleveDao" class="org.lilie.socle.api.annuaire.dao.impl.EleveDaoImpl"*

*parent="annuaireParentDao"/>*

**

### Exceptions

La gestion des exceptions à mettre en place est définie dans son ensemble dans la spécification technique du fmk-core-ent.

### Utilisation d’Ibatis

Les DAO accédant à la base de données, doivent être développés avec Ibatis afin d’être homogènes avec le reste de l’ENT. Ibatis permet entre autres de réaliser plus ou moins automatiquement un mapping entre les tables et les objets Java, et ainsi de simplifier les accès.

C’est l’objet sqlMapClient de la classe mère qui permet d’utiliser les méthodes d’Ibatis :

* *sqlMapClient.queryForObject(String nomRequete, Object param)*

Cette méthode permet d’appeler une requête en consultation, en fournissant :

* + le namespace+nom de la requête dans le fichier de configuration (cf paragraphe suivant)
  + le(s) paramètre(s) : un type simple, un objet (dans ce cas dans la requête on utilise directement les noms des attributs de l’objet), ou une map (dans la requête on utilise les clés)
* Les méthodes *insert()*, *update()* et *delete()* permettent de faire des accès en écriture.

Spécificité ENT :

Lors du passage d’un unique paramètre, le nom correspondant utilisé dans la requête ensuite peut être n’importe lequel. Cependant, afin d’éviter les erreurs de lecture, il vaut mieux utiliser dans ce cas là toujours le même. Dans l’ENT, nous utilisons : *#value#.*

Exemple d’erreur possible en maintenance sinon : on commence par passer le nom de la personne, on nomme donc le paramètre *#nom#* dans la requête. Finalement la méthode change et on décide de passer l’identifiant. On change donc la valeur passée en paramètre mais on oublie de changer le *#nom#* en *#id#* dans la requête. Cela continuera de fonctionner mais on aura un problème de cohérence : on passe l’id mais on lit *#nom#* à la place du paramètre. Cela peut prêter à confusion lors de la maintenance et du débuguage.

Pour le passage de plusieurs paramètres, la question ne se pose pas.

### Définition des requêtes dans les SqlMap

Ibatis fonctionne avec des fichiers xml de configuration. On les appelle des *sqlMap*. Dans l’ENT, on en crée un différent pour chaque DAO, afin de retrouver facilement les requêtes correspondantes par fonctionnalité : *<fonctionnalite>SQLMap.xml,*

placé dans *src/main/resources*, dans une arborescence de dossier similaire au package de la classe Java, plus un sous-dossier « *map* ».

Chaque fichier SQLMap doit être déclaré dans le fichier chargé par Spring :

*src/main/resources/SqlMapConfig.xml*.

*<sqlMapConfig>*

*<settings useStatementNamespaces="true"/>*

*<sqlMap resource="org/lilie/socle/api/annuaire/dao/map/EleveSQLMap.xml"/>*

*…*

*</sqlMapConfig>*

Contenu du SqlMap :

* La balise sqlMap contient un attribut *namespace* qui permet de définir un nom et ainsi d’indiquer quel fichier de configuration on utilise lors de l’appel de la requête : on utilise le nom de la fonctionnalité déjà utilisée dans le nom du fichier.
* Balise *typeAlias* : on peut créer des alias sur des types d’objets Java afin de ne pas avoir à répéter le package sans arrêt dans le fichier lors de l’utilisation.
* Balise *resultMap* : elle permet de définir un mapping entre les colonnes d’une table et les attributs d’un objet Java (utilisation de l’alias).

Par convention, on crée un *resultMap* pour chaque type d’objet, sans les jointures ; puis un autre *resultMap* qui étend le premier (attribut *extends* de la balise) pour les jointures voulues.

* Les balises *select/insert/update/delete* (avec un chacune un attribut *id* qui permet l’appel depuis le dao) permettent de définir des requêtes. Plusieurs balises Ibatis de logique peuvent être utilisées (conditions, itérations…), ainsi que des *include* d’autres fragments de sql définis dans ce sqlMap ou un autre…

Ce sont ces balises qui utilisent les paramètres (#value#) passés comme vu plus haut. Le *resultMap* à utiliser pour cette requête est à préciser également. Par contre il ne sert que pour les requêtes de type *select*.

Exemple de fichier sqlMap :

*<sqlMap namespace="eleve">*

*<typeAlias alias="eleveBean" type="org.lilie.socle.api.annuaire.dao.pojo.Eleve"/>*

*<resultMap id="dbEleveResult" class="eleveBean" extends="personne.dbPersonneResult" groupBy="personneId">*

*<result column="ville\_naissance" property="villeNaissance"/>*

*<result column="departement\_naissance" property="departementNaissance"/>*

*<result column="pays\_naissance" property="paysNaissance"/>*

*<result column="boursier" property="boursier"/>*

*<result column="statut" property="statut"/>*

*<result column="delegue\_classe" property="delegueClasse"/>*

*<result column="delegue\_autres" property="delegueAutres"/>*

*<result column="majeur" property="majeur"/>*

*<result column="majeur\_anticipe" property="majeurAnticipe"/>*

*<result column="transport" property="transport"/>*

*<result column="mef\_rattachement" property="mefRattachement"/>*

*<result column="mef\_id" property="mef.mefId"/>*

*<result column="niveau\_formation" property="niveauFormation.id"/>*

*<result column="filiere\_id" property="filiere.id"/>*

*<result column="regime\_id" property="regime.id"/>*

*<result column="specialite" property="specialite.id"/>*

*<result column="id\_sconet" property="idSconet"/>*

*</resultMap>*

*<resultMap id="dbEleveResultAvecEtab" class="eleveBean" extends="dbEleveResult">*

*<result property="etablissement" resultMap="etablissement.dbEtablissementResult" />*

*</resultMap>*

*<select id="getEleveParId" parameterClass="long" resultMap="dbEleveResult">*

*SELECT*

*personne\_id,*

*jointure,*

*id\_sconet,*

*etablissement\_rattachement\_id as etab\_etablissement\_id,*

*nom\_canonique,*

*nom\_usage,*

*mot\_passe,*

*telephone\_professionnel,*

*titre,*

*fax,*

*nom\_affichage,*

*prenom,*

*telephone\_personnel,*

*photo,*

*email,*

*telephone\_portable,*

*numero\_bureau,*

*prenoms\_autre,*

*nom\_patronymique,*

*adresse\_personnelle,*

*code\_postal\_personnelle,*

*ville\_personnelle,*

*pays\_personnelle,*

*login,*

*alias,*

*date\_naissance,*

*sexe,*

*civilite,*

*PERSO\_DATA\_VISIBLE\_ELEVE,*

*PERSO\_DATA\_VISIBLE\_ENSEIGNANT,*

*PERSO\_DATA\_VISIBLE\_PERSRELELEVE,*

*PERSO\_DATA\_VISIBLE\_NON\_ENSEIGN,*

*PERSO\_DATA\_VISIBLE\_INVITE,*

*COMPTE\_INITIALISE,*

*QUESTION\_SECRETE,*

*REPONSE\_SECRETE,*

*DATE\_FIN\_BLOCAGE\_COMPTE,*

*DERNIERE\_CONNEXION,*

*CONNEXION,*

*MOT\_PASSE\_REINITIALISE,*

*DATE\_SUPPRESSION,*

*DATE\_SUSPENSION,*

*ville\_naissance,*

*departement\_naissance,*

*pays\_naissance,*

*boursier,*

*statut,*

*mef\_rattachement,*

*niveau\_formation\_diplome,*

*specialite,*

*delegue\_classe,*

*delegue\_autres,*

*majeur,*

*majeur\_anticipe,*

*mef\_id,*

*transport,*

*filiere\_id,*

*regime\_id,*

*niveau\_formation,*

*<include refid="selectProfilEleve" />*

*FROM*

*enteleve*

*WHERE*

*personne\_id = #value#*

*</select>*

*…*

*</sqlMap>*

## Configuration de la datasource

On définit systématiquement deux beans pour la connexion à la base de données : un bean qui pointe vers la datasource JNDI du Tomcat, un autre qui pointe vers les propriétés de connexion définies directement dans le fichier *config.properties*. Exemple de l’api-annuaire :

*<bean id="dataSourceJNDIAnnuaire" class="org.springframework.jndi.JndiObjectFactoryBean" lazy-init="true">*

*<property name="jndiName" value="$annuaire{DATABASE\_JNDI}"/>*

*</bean>*

*<bean id="dataSourceAnnuaire" class="org.postgresql.ds.PGPoolingDataSource" lazy-init="true">*

*<property name="serverName" value="$annuaire{DATABASE\_SERVER\_NAME}"/>*

*<property name="portNumber" value="$annuaire{DATABASE\_PORT\_NUMBER}"/>*

*<property name="databaseName" value="$annuaire{DATABASE\_NAME}"/>*

*<property name="user" value="$annuaire{DATABASE\_USER}"/>*

*<property name="password" value="$annuaire{DATABASE\_PASSWORD}"/>*

*<property name="dataSourceName" value="$annuaire{DATABASE\_DATASOURCE\_NAME}"/>*

*<property name="initialConnections" value="$annuaire{DATABASE\_INITIAL\_CONNECTIONS}"/>*

*<property name="maxConnections" value="$annuaire{DATABASE\_MAX\_CONNECTIONS}"/>*

*</bean>*

Le *lazy-init* permet de ne charger ces beans que lorsqu’ils sont réellement utilisés (si le JNDI n’existe pas, cela ne pose ainsi pas de problème tant qu’il n’est pas utilisé).

Afin de choisir quel type de définition de datasource est à utiliser, les modules utilisant des apis ou une base de données spécifique doivent préciser la propriété *DATABASE\_DEFINITION\_TYPE* dans le *config.properties* du module. La valeur doit être à « *dataSourceJNDI »* pour utiliser les datasources JNDI définies dans Tomcat. Sinon avec « *dataSource* » ce sont les datasources définies dans le contexte Spring qui sont utilisées (par exemple sur les postes de développement ou pour les batchs). C’est bien le fichier de config du module qui doit contenir la propriété, car c’est lui (webapp ou batch) qui porte le choix. Ce choix est ensuite appliqué à la fois à la base éventuelle du module et à celles des apis.

Configuration Ibatis :

*<bean id="annuaireSqlMap" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">*

*<property name="configLocation">*

*<value>classpath:SqlMapConfig\_api-annuaire.xml</value>*

*</property>*

*<property name="dataSource" ref="${DATABASE\_DEFINITION\_TYPE}Annuaire" />*

*</bean>*

## Utilisation de POJO

Les objets Java utilisés par les DAO sont des POJO, c'est-à-dire de simples beans composés d’attributs et de leurs accesseurs. Ceux-ci sont mappés aux tables de la base de données grâce à des fichiers de configuration sqlMap. Ils collent au modèle de données et en reflètent les changements.

Du coup ils ne doivent être utilisés que dans la partie back du projet. D’autres objets, plus proches de la vision métier que de la vision peristance, sont utilisés par les couches au dessus. Ceci permet de bien isoler les couches.

Les POJO se trouvent dans un sous-package « *pojo* » du package « *dao* ». Ils doivent tous hériter du *Pojo* de référence du package « *reference* » du projet, qui pointe lui-même vers celui du core.

# Développement de la couche métier

## Présentation

Les classes métiers permettent de définir les traitements fonctionnels, les règles de gestion. Ils peuvent faire appel à des DAO, à des API spécifiques, ou à d’autres classes métiers. Contrairement aux DAO qui se limitent à des traitements unitaires, les services métiers regroupent un certains nombre de traitements représentant une logique fonctionnelle, comme par exemple l’appel à plusieurs DAO pour récupérer différentes données liées fonctionnellement.

Afin de pouvoir évoluer vers une autre solution d’accès, pour chaque métier on utilise des interfaces permettant de ne montrer que la signature des méthodes disponibles en cachant l’implémentation réelle à la couche supérieure.

Le lien entre la couche supérieure et l’implémentation est fait par l’utilisation du concept d’IoC de Spring : la couche supérieure ne crée pas directement l’objet représentant la couche inférieure qu’elle utilise, c’est Spring qui s’en charge par injection. Cela permet par exemple de changer facilement l’implémentation utilisée, la couche supérieure ne se souciant que de ce que lui fournit l’interface.

## Implémentation des classes métiers

### Classes de référence

Le package pour la couche métier est le suivant :

*org.lilie.socle.api.<projet>.business* pour une api

*org.lilie.services.<module>.business* pour un module

*org.lilie.socle.bat.<projet>.business* pour un batch

Un sous-package « reference » permet de mettre les différentes classes de référence du projet, c'est-à-dire les classes mères de la couche métier, qui servent de « modèles ».

La classe de référence *Business* de chaque projet hérite de celle de *fmk-core-ent*. Elle permet de centraliser les traitements communs éventuels des business du projet.

L’attribut *adapter* utilisé par la majorité des business n’y est pour l’instant pas centralisé, mais pourrait l’être, avec également un bean spring parent correspondant, pour éviter d’avoir à répéter cette propriété dans chaque business et chaque bean (cf comme pour l’attribut *sqlMapClient* du Dao).

### Héritage et nomenclatures

On crée en général une classe métier par fonctionnalité ou type de données. Il est possible également d’ajouter un découpage en plusieurs sous-packages, afin d’organiser en différents domaines si les classes sont nombreuses.

Comme précisé au-dessus, le métier est composé de deux fichiers, l’interface et l’implémentation. L’interface doit se nommer *<fonctionnalite>Business.java*  et être dans la package racine « *business* ». L’implémentation doit se nommer *<fonctionnalite>BusinessImpl.java*, ou *<fonctionnalite>BusinessJerseyImpl.java* par exemple si l’on veut préciser le type d’implémentation. Les implémentations se mettent dans un sous-package « *impl* ».

L’interface ne contiendra que la signature des méthodes déclarées *public* devant être disponibles pour la couche supérieure. Les méthodes d’implémentation n’étant utilisées qu’en local doivent être déclarées en *private* dans la classe d’implémentation.

La classe d’implémentation doit faire un *implements* explicite de son interface.

Il faut déclarer chaque business dans la configuration Spring afin qu’un bean correspondant soit disponible pour les couches supérieures qui veulent l’utiliser. Dans le fichier *business-applicationContext.xml*, on ajoute l’élément suivant (exemple pour l’api-annuaire):

*<bean id="eleveBusiness" class="org.lilie.socle.api.annuaire.business.impl.EleveBusinessImpl">*

C’est cette ligne qui fait le lien entre l’interface appelée par le métier et l’implémentation réellement utilisée. Le nom du bean doit reprendre celui de l’interface et non celui de l’implémentation afin de garder les couches étanches.

Chaque implémentation doit également étendre la classe mère *Business* du projet afin d’hériter des fonctionnalités communes.



### Appel des autres couches

Pour appeler un Dao ou un autre business, il faut que celui-ci soit chargé par Spring par injection :

* Ajout d’un attribut avec le type de l’interface du Dao
* Ajout du setter de cet attribut
* Ajout de cet attribut dans la définition du bean Spring (sans jamais préciser explicitement l’implémentation, mais en utilisant le nom du bean Spring défini dans les autres fichiers applicationContext.xml).

*<bean id="eleveBusiness" class="org.lilie.socle.api.annuaire.business.impl.EleveBusinessImpl">*

*<property name="referentielBusiness" ref="referentielAnnuaireBusiness" />*

*<property name="personneBusiness" ref="personneBusiness" />*

*<property name="relationDao" ref="relationDao" />*

*<property name="eleveDao" ref="eleveDao" />*

*<property name="adapter" ref="annuaireAdapter" />*

*</bean>*

### Gestion des contrôles fonctionnels

La gestion des exceptions à mettre en place est définie dans son ensemble dans la spécification technique du fmk-core-ent. Nous ne reprendrons donc pas ici la façon dont elles doivent être catchées.

Voici par contre ce qu’il y a à faire dans le cas d’un contrôle fonctionnel :

C’est la couche métier qui est responsable des traitements fonctionnels. Les règles de gestion doivent donc être contrôlées dans le corps des méthodes des business. Lorsqu’un contrôle n’est pas valide, on doit alors lancer une nouvelle exception de type *ServiceFonctionnelleException,* avec un code erreur fonctionnel correspondant au contrôle effectué :

*throw new ServiceFonctionnelleException(codeErreur, message) ;*

Les codes erreur (stockés dans des constantes Java), sont connus des couches supérieures qui peuvent ainsi réagir en fonction pour par exemple afficher un message à l’utilisateur.

Le message fournit en paramètre du constructeur est un message par défaut pour les logs techniques. Ce n’est pas celui qui devrait être utilisé pour l’affichage dans l’interface de l’utilisateur, puisque ce n’est pas à la couche métier de définir les libellés pour le front.

### Appel de Web Services

Si la couche métier a besoin de faire des appels web services plutôt que des appels de DAO classiques, il faut dans ce cas bien les encapsuler de manière à pouvoir s’en détacher :

* Utilisation de la librairie Axis dont la version est commune et déclarée dans le pom parent ent.
* Les WSDL sont à mettre dans *src/main/resources/wsdl.*
* La génération des classes Java associées est à faire dans la configuration Maven :

*<properties>*

*<dir.wsdl>${basedir}/src/main/resources/wsdl</dir.wsdl>*

*</properties>*

*<build>*

*<plugins>*

*<!-- Generation du java a partir des wsdl -->*

*<plugin>*

*<groupId>org.codehaus.mojo</groupId>*

*<artifactId>axistools-maven-plugin</artifactId>*

*<configuration>*

*<sourceDirectory>${dir.wsdl}</sourceDirectory>*

*<wsdlFiles>*

*<wsdl>monfichier.wsdl</wsdl>*

*</wsdlFiles> <packageSpace>org.lilie.services.nommodule.ws*

*</packageSpace>*

*<verbose>true</verbose>*

*<allElements>true</allElements>*

*<testCases>false</testCases>*

*</configuration>*

*<executions>*

*<execution>*

*<phase>process-resources</phase>*

*<goals>*

*<goal>wsdl2java</goal>*

*</goals>*

*</execution>*

*</executions>*

*</plugin>*

*<plugins>*

*<build>*

Le package de destination des classes Java (*packageSpace*) doit être un package nommé « *ws* » à part par rapport aux autres couches.

* L’utilisation des classes générées par Axis doit être encapsulée dans une ou plusieurs classes business neutres à part regroupées comme une sorte d’api dans le projet. Les signatures des méthodes ne doivent pas refléter que ce sont des appels ws. Les différents business du module peuvent ensuite appeler cette « api ».
* Si ces web services peuvent être utiles à d’autres modules, il ne faut alors pas les laisser dans le projet du module. Dans ce cas, on crée :
  + un projet *ws-fonctionnalite* au sein du socle, qui contient les wsdl et la génération des classes Java.
  + un projet *api-fonctionnalite* au sein du socle, qui contient l’encapsulation des appels aux classes Java ws dans des business neutres. Cette api a une dépendance sur le projet *ws-fonctionnalite*.
  + Le module a seulement une dépendance sur le projet *api-fonctionnalite* sans être conscient des appels ws*.*

## Les objets métiers

### DTO

Les POJO utilisés par les DAO sont générés automatiquement et sont très liés au modèle de la base de données. Pour le métier et le front, on a besoin d’objets plus fonctionnels ou plus proches de la représentation utilisateur dans la vue. La couche présentation ne doit pas dépendre du fonctionnement du back. On utilise donc d’autres objets Java, les DTO, pour transporter les données. Ceux-ci sont également de simples beans avec des accesseurs, mais ils peuvent traverser les couches car ils ne sont pas dépendants du modèle physique.

Ils seront utilisés à la fois dans les services métiers, les contrôleurs et la vue. Ils doivent donc se trouver dans le package :

*org.lilie.socle.api.<projet>.dto* pour une api

*org.lilie.services.<module>.dto* pour un module

*org.lilie.socle.bat.<projet>.dto* pour un batch.

Ils doivent tous hériter de la classe de référence *Dto* dans le sous-package « reference », qui hérite elle-même de celle du core. Elle rend notamment les dto sérialisables.

Les DTO doivent être nommés *<fonctionnalité>Dto*. S’ils correspondent en particulier à un objet POJO de la couche DAO, on reprend le même nom de fonctionnalité.

Exemple :

Pojo : *Personne.java*

Dto : *PersonneDto.java*

### Adapter

Dans les services métiers et dans le front, on utilise des DTO. Or le métier fait appel à des DAO qui utilisent des POJO. Il faut donc faire le lien entre ces deux types de données afin de pouvoir passer les informations de l’un à l’autre. Pour cela, on doit faire un mapping entre chaque DTO et son ou ses POJO correspondants. Au lieu de faire cela manuellement dans chaque méthode, un adapter peut s’en charger automatiquement pour les cas les plus simples.

Nous utilisons Dozer. Cet outil permet de mapper facilement différents objets Java :

* Si les attributs ont le même nom et le même type, ils seront automatiquement mappés, pas besoin de les préciser dans la configuration.
* Les listes, maps et tableaux sont mappés de l’une vers l’autre automatiquement.
* On peut surcharger et customiser un mapping : par exemple, grâce aux customisations faites dans fmk-core-ent (cf documentation), les booléens true/false des DTO peuvent être mappés avec les String O/N du modèle.

*<configuration>*

*<custom-converters>*

*<converter type="org.lilie.socle.core.utils.adapter.ConverterBooleanString" >*

*<class-a>java.lang.String</class-a>*

*<class-b>java.lang.Boolean</class-b>*

*</converter>*

*<converter type="org.lilie.socle.core.utils.adapter.ConverterDateString">*

*<class-a>java.lang.String</class-a>*

*<class-b>java.util.Date</class-b>*

*</converter>*

*<converter type="org.lilie.socle.core.utils.adapter.ConverterStringDouble">*

*<class-a>java.lang.String</class-a>*

*<class-b>java.lang.Double</class-b>*

*</converter>*

*<converter type="org.lilie.socle.core.utils.adapter.ConverterStringInteger">*

*<class-a>java.lang.String</class-a>*

*<class-b>java.lang.Integer</class-b>*

*</converter>*

*<converter type="org.lilie.socle.core.utils.adapter.ConverterStringString">*

*<class-a>java.lang.String</class-a>*

*<class-b>java.lang.String</class-b>*

*</converter>*

*</custom-converters>*

*</configuration>*

* Dozer fonctionne également en cascade sur les objets imbriqués.
* Des fichiers de configuration xml permettent de mapper les champs n’ayant pas le même nom.

Exemple de configuration :

*<mapping>*

*<class-a>org.lilie.socle.api.annuaire.dto.EleveDto</class-a>*

*<class-b>org.lilie.socle.api.annuaire.dao.pojo.Eleve</class-b>*

*<field>*

*<a>mefDto</a>*

*<b>mef</b>*

*</field>*

*<field>*

*<a>niveauFormationDto</a>*

*<b>niveauFormation</b>*

*</field>*

*<field>*

*<a>filiereDto</a>*

*<b>filiere</b>*

*</field>*

*<field>*

*<a>regimeDto</a>*

*<b>regime</b>*

*</field>*

*<field>*

*<a>niveauFormationDiplomeDto</a>*

*<b>niveauFormationDiplome</b>*

*</field>*

*<field>*

*<a>specialiteDto</a>*

*<b>specialite</b>*

*</field>*

*<field>*

*<a>relClasseEleveDtoList</a>*

*<b>relClasseEleveList</b>*

*</field>*

*</mapping>*

Le fichier de configuration est à mettre dans : *src/main/resources/dozer/dozerMappings.xml*.

Il faut également déclarer le bean Spring associé dans *adapter-applicationContext.xml* :

*<bean id="adapter"*

*class="org.lilie.socle.core.utils.adapter.AdapterDozerImpl" singleton="true">*

*<property name="mappingFiles">*

*<list>*

*<value>dozer/dozerMappings.xml</value>*

*</list>*

*</property>*

*</bean>*

L’utilisation de Dozer dans le métier se fait de la manière suivante :

* Utilisation de l’objet *Adapter* de fmk-core-ent
* Déclaration de l’attribut pour l’injection Spring, comme dit plus haut pour les Dao
* Deux méthodes permettent de faire la transformation :
  + *adapter.mapToList(personneList, PersonneDto.class)* renvoie une liste de *PersonneDto*, *personneList* étant une liste de POJO Personne
  + *adapter.map(personne, PersonneDto.class)* qui est l’équivalent pour un objet simple

Le mapping se fait dans les deux sens :

* transformation des paramètres DTO en POJO pour les passer au DAO
* transformation des résultats POJO du DAO en DTO pour les remonter à la vue

## Gestion du cache

Si un business nécessite une mise en cache de certaines données, la mise en place de celui-ci est décrite dans la documentation sur le fmk-core-ent.

# Transactions

TODO

# Constantes et utilitaires

En plus des packages *dao* et *business*, chaque projet possède un package *utils* qui contient tous les utilitaires et constantes.

Les constantes sont dans le fichier *Constantes.java*. Les *Enum* du projet sont également à placer dans le package *utils*.

# Annexe

Liste non exhaustive des points vérifiés lors des revues de code :

|  |
| --- |
|  |
| Architecture du projet |
| répartition en couches |
| utilisation d'interfaces |
| utilisation des classes de référence |
| utilisation de POJOs |
| gestion des exceptions |
| utilisation du logger ent |
| découpage fonctionnel (homogène entre les couches) |
| glissements de couche |
| Utilisation des utilitaires du socle |
| gestion des fichiers et images |
| gestion des droits |
| gestion de l'utilisateur connecté |
| gestion des fichiers de conf (config et log4j) |
| gestion de la navigation: choixTraitementStr, provenanceStr |
| utilisation d'ajax |
| Organisation des fichiers de configurations |
| noms et organisation des fichiers de config spring |
| noms et organisation des fichiers de config dozer |
| noms et organisation des sqlmap |
| pom.xml |
| Utilisation des apis |
| passage obligatoire par api-portail pour webapps |
| imports des contextes spring (sans doublons) |
| Normes de nommage |
| noms des méthodes |
| noms des variables et attributs |
| noms des constantes |
| noms des beans spring |
| noms des sqlmap |
| noms des requêtes des sqlmap |
| noms des pojos et dtos |
| noms des classes DAO |
| noms des classes Business |
| noms des classes actions |
| noms des paths struts des actions |
| noms des jsp et js |
| noms des clés dans le config.properties |
| noms des clés et organisation des messages.properties |
| Cosmétique |
| commentaires javadoc de classe |
| commentaires javadoc de méthodes |
| commentaires de blocs |
| commentaires des getter-setter (auto eclipse) |
| commentaires des config.properties |
| commentaires js |
| utilisation des css de fmk-core-web |
| commentaire de licence ent |
| Couche persistance |
| utilisation d'un namespace unique par sqlmap |
| utilisation du nom "value" pour un param unique dans le sqlmap |
| bonne utilisation d'ibatis |
| Couche business |
| gestion des RG: exceptions fonctionnelles + codes erreurs |
| bonne utilisation de dozer |
| organisation des dtos |
| Couche contrôleur |
| utilisation des Init et Fin |
| pas de méthode "execute" |
| répartition des méthodes dans les différentes classes |
| utilisation du validator struts |
| bonne utilisation des conversations (pas tous dans la même conv, utilisation du POST qd c'est possible) et de la session |
| pas de mapping inutile, utilisation dtos |
| sécurité des actions (utilisation des aspects) |
| Couche présentation |
| organisation des jsp (include jspf, ajax, nombre de jsp…) |
| externalisation du javascript |
| indentation dans les jsp et js |
| utilisation de sitemesh |
| jspf pour les template sitemesh dans "decoration" |
| utilisation de displaytag |
| pas de libellés en dur |
| pas d'url en dur (<s:url>) |
| utilisation du POST et non du GET |
| utilisation des balises struts plutôt que html |
| imports en double (avec decorator par ex) |
| Algorithmie, optimisations, codes à risque… |
| organisation des méthodes: longueur, factorisation |
| attention aux chargements en mémoire (méthodes trop longues à découper etc) |
| optimisations: StringBuilder, for(pasDeMethode), utilisation de variable à bon escient |
| attention aux NullPointerException (test de nullité) |
| if(boolean) avec return boolean : test inutile |
| Enum: utiliser des ==, des switch (pas des equals et des if) |
| Divers |
| checkstyle |
| bonne utilisation des niveaux de trace |
| déclaration du logger avec la bonne classe |
| pas de constantes en dur (string, entiers ou clés de properties) |
| utilisation d'objets typés (pas Object) et Long/Integer plutôt que long/int |
| scope=singleton sauf pour actions scope=prototype |
| supprimer les bouts de codes et classes inutilisés |
| lisibilité générale |