**Spécification technique – Frameworks core**

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

**de la région Île-de-France**

**Spécification Technique  
Frameworks core**

**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 | 23/05/2011 | MMAU | Initialisation du document |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Sommaire

[Introduction 5](#_Toc299371091)

[1. Fmk-core-ent 5](#_Toc299371092)

[1.1. Classes de référence 5](#_Toc299371093)

[1.2. Gestion des exceptions 6](#_Toc299371094)

[1.2.1. Classes du core 6](#_Toc299371095)

[1.2.2. Utilisation 7](#_Toc299371096)

[1.2.3. Logger 9](#_Toc299371097)

[1.3. Mapping POJO 9](#_Toc299371098)

[1.4. Conteneur Spring 9](#_Toc299371099)

[1.5. Gestion des fichiers de configuration des projets 9](#_Toc299371100)

[1.5.1. Objectif 9](#_Toc299371101)

[1.5.2. Utilisation pour un module 10](#_Toc299371102)

[1.5.3. Fonctionnement interne 13](#_Toc299371103)

[1.6. Gestion du cache 14](#_Toc299371104)

[1.6.1. Quand utiliser du cache 14](#_Toc299371105)

[1.6.2. Comment l’utiliser 14](#_Toc299371106)

[1.7. Gestion de l’utilisateur connecté 15](#_Toc299371107)

[1.8. Echanges de données entre modules 15](#_Toc299371108)

[1.8.1. Communication possible entre modules 15](#_Toc299371109)

[1.8.2. Contrat entre les deux modules 15](#_Toc299371110)

[1.8.3. Utilisation par un module appelant 16](#_Toc299371111)

[1.8.4. Mise à disposition de ressources par un module 17](#_Toc299371112)

[1.8.5. Cas particulier des ressources de notifications 19](#_Toc299371113)

[1.8.6. Fonctionnement interne 20](#_Toc299371114)

[1.9. Gestion des fichiers uploadés 21](#_Toc299371115)

[1.9.1. Objectif 21](#_Toc299371116)

[1.9.2. Sécurisation 21](#_Toc299371117)

[1.9.3. Discriminant d’un fichier 21](#_Toc299371118)

[1.9.4. Upload d’un fichier 21](#_Toc299371119)

[1.9.5. Récupération d’un fichier stocké 22](#_Toc299371120)

[1.9.6. Fonctionnement interne 22](#_Toc299371121)

[1.10. Encapsulation d’api FTP 23](#_Toc299371122)

[1.10.1. Description générale 23](#_Toc299371123)

[1.10.2. Fonctionnement interne 23](#_Toc299371124)

[1.11. Autres utilitaires divers 24](#_Toc299371125)

[1.11.1. Outil de cryptage 24](#_Toc299371126)

[1.11.2. Email 24](#_Toc299371127)

[1.11.3. Libellés 24](#_Toc299371128)

[1.11.4. Constantes 24](#_Toc299371129)

[1.12. Configuration du core 24](#_Toc299371130)

[1.13. Packaging du projet 24](#_Toc299371131)

[2. Fmk-core-web 25](#_Toc299371132)

[2.1. Présentation 25](#_Toc299371133)

[2.2. Classes de référence 25](#_Toc299371134)

[2.3. Gestion de l’utilisateur connecté 25](#_Toc299371135)

[2.4. Gestion des données en session 26](#_Toc299371136)

[2.4.1. Objectif 26](#_Toc299371137)

[2.4.2. Utilisation des conversations 26](#_Toc299371138)

[2.4.3. Fonctionnement interne 26](#_Toc299371139)

[2.5. Gestion des erreurs 27](#_Toc299371140)

[2.5.1. Gestion dans le contrôleur 27](#_Toc299371141)

[2.5.2. Pages d’erreur 27](#_Toc299371142)

[2.5.3. Page de supervision 28](#_Toc299371143)

[2.6. Export 28](#_Toc299371144)

[2.6.1. Utilisation 28](#_Toc299371145)

[2.6.2. Fonctionnement 28](#_Toc299371146)

[2.7. Gestion des fichiers uploadés 28](#_Toc299371147)

[2.7.1. Contrôleur 28](#_Toc299371148)

[2.7.2. Utilitaires pour la vue 29](#_Toc299371149)

[2.7.3. Fonctionnement interne 30](#_Toc299371150)

[2.8. Ressources statiques communes 30](#_Toc299371151)

[2.8.1. Objectif 30](#_Toc299371152)

[2.8.2. Ressources pour le look 30](#_Toc299371153)

[2.8.3. Utilitaires Javascript et JSP 31](#_Toc299371154)

[2.8.4. Autres 31](#_Toc299371155)

[2.9. Configuration 31](#_Toc299371156)

[2.10. Packaging 31](#_Toc299371157)

[2.10.1. Dépendances 31](#_Toc299371158)

[2.10.2. Déploiement des ressources web 32](#_Toc299371159)

[2.10.3. Récupération des ressources web par les webapps 33](#_Toc299371160)

# Introduction

Le but de ce document est de décrire le fonctionnement et l’utilisation des projets core du socle de l’ENT. Comme dit dans la documentation sur l’ensemble du socle applicatif, il y a deux projets core principaux utilisés par l’ensemble des projets ENT :

* fmk-core-ent : à utiliser pour tous les projets ENT, aussi bien les modules que les apis et batchs du socle
* fmk-core-web : à utiliser en plus du précédent pour les webapps en client léger

Ce sont des frameworks spécifiques ENT qui ont plusieurs rôles :

* Donner une couche d’abstraction des outils techniques utilisés
* Proposer un socle technique de départ pour les nouveaux projets
* Centraliser les traitements qui peuvent être commun à tous les projets

L’utilisation et les configurations communes avec les autres projets du socle sont décrites dans la documentation du socle applicatif. Seules les particularités des deux projets sont détaillées ici.

# Fmk-core-ent

## Classes de référence

Le projet est divisé en couches de la même manière que tous les projets ENT. Dans le package de chaque couche, on retrouve les classes de référence pour chacune. Les classes des différentes applications et apis doivent hériter de ces classes de référence dans lesquelles on peut centraliser des traitements communs.

* *org.lilie.socle.core.business.Business*
* *org.lilie.socle.core.dao.IbatisDao*

Cette classe centralise la référence vers l’objet sqlMapClient utilisé pour les requêtes Ibatis.

* *org.lilie.socle.core.dao.pojo.Pojo*

Tous les POJO héritant de cette classe seront serialisables.

* *org.lilie.socle.core.dto.Dto*

Tous les DTO héritant de cette classe seront serialisables.

## Gestion des exceptions

### Classes du core

Le core propose également des objets exceptions spécifiques aux ENT afin de maîtriser la gestion des erreurs remontées.

* Toutes les exceptions ENT héritent de la classe :

*org.lilie.socle.core.utils.exception.Exception*

* Les exceptions sont gérées par couche : Il existe donc des exceptions de type *Service* pour les business et de type *Dao* pour les dao. Elles sont dans les packages correspondants.
* Deux types d’exceptions pour chaque couche :
  + les exceptions techniques : erreurs techniques non prévues. Cas anormal à traiter de manière générique.
  + les exceptions fonctionnelles : erreurs fonctionnelles prévues par l’application, règles de gestion… Cas fonctionnel normal géré par l’application, pour lequel on veut en général afficher un message à l’utilisateur. Un code d’erreur doit être précisé pour les exceptions fonctionnelles afin que le front puisse ensuite décider de la marche à suivre selon l’erreur.

Chaque couche gère son propre type d’exception en encapsulant les exceptions des couches d’en-dessous. Les exceptions des couches les plus basses ne doivent pas être remontées directement jusqu’en haut, mais doivent être tracées puis transformées à chaque couche. Cela permet de garder une étanchéïté entre les couches. Chacune gère ce qu’elle connaît et se cantonne à son propre travail. Les exceptions remontent ainsi jusqu’au front, qui décide ce qu’il doit en faire selon le type fonctionnel ou technique de l’erreur.

* Il existe également deux autres types d’exception transverses dans le package *org.lilie.socle.core.utils.exception* :
  + *ParserTechniqueException* : exception de type technique, utilisée lors d’une erreur de parsing xml. Cela permet de préciser dans ce cas là le type d’exception technique.
  + *AccesNonAutoriseException*: exception envoyée par les traitements de gestion de la sécurisation des urls et des données lors d’un accès non autorisé à une fonctionnalité.

### Utilisation

Chaque couche ne doit renvoyer que des exceptions qui lui sont propres. Elle doit donc rattraper, tracer et transformer chaque exception intervenue dans le corps de chacune de ses méthodes.

La couche la plus basse, le dao, ne doit renvoyer que des exceptions de type Dao, et donc avoir dans sa signature :

*throws DaoFonctionnelleException, DaoTechniqueException*

Elle rattrape toutes les exceptions techniques possibles, les trace en mode ERROR et les renvoie en tant que *DaoTechniqueException* :

*try {*

*…*

*} catch (Exception e) {*

*LOGGER.error(e.getMessage(), e);*

*throw new DaoTechniqueException(e.getMessage(), e);*

*}*

La couche business utilise les méthodes du dao et doit donc gérer les exceptions renvoyées par celle-ci. Elle doit donc rattraper les exceptions de type *Dao*, mais aussi les exceptions autres générées par le corps de la méthode business hors appel du dao. Tout cela doit ensuite être transformé en exception de type *Service*, afin que la méthode puisse avoir comme signature :

*throws ServiceTechniqueException, ServiceFonctionnelleException*

Les exceptions *DaoTechnique* sont transformées en *ServiceTechnique*, les *DaoFonctionnelle* sont transformées en *ServiceFonctionnelle* :

*try {*

*…*

*} catch (DaoFonctionnelleException dfe) {*

*LOGGER.info(dfe.getMessage());*

*throw new ServiceFonctionnelleException(dfe.getErrorCode(), dfe.getMessage());*

*} catch (DaoTechniqueException dte) {*

*LOGGER.error(dte.getMessage());*

*throw new ServiceTechniqueException(dte.getMessage(), dte);*

*} catch (Exception e) {*

*LOGGER.error(e.getMessage(), e);*

*throw new ServiceTechniqueException(e.getMessage(), e);*

*}*

Les exceptions fonctionnelles sont tracées en INFO. Les exceptions rattrapées pour la première fois par l’application (bloc Exception gérant les exceptions ne venant pas des couches inférieures) doivent être tracées en fournissant la référence vers l’objet erreur, afin de logger la *stacktrace* de l’erreur. Les fois d’après, quand l’exception est rattrapée par les autres couches au-dessus, on ne fournit plus l’objet au logger.

Attention, quand une méthode business utilise une autre méthode d’un business, il faut penser à rattraper également explicitement les erreurs correspondantes. Sinon elles seront enterrées en étant rattrapées comme des exceptions génériques *Exception*. Des exceptions *ServiceFonctionnelle* pourraient ainsi se retrouver enterrées en *ServiceTechnique*.

*} catch (DaoFonctionnelleException dfe) {*

*LOGGER.info(dfe.getMessage());*

*throw new ServiceFonctionnelleException(dfe.getErrorCode(), dfe.getMessage());*

*} catch (DaoTechniqueException dte) {*

*LOGGER.error(dte.getMessage());*

*throw new ServiceTechniqueException(dte.getMessage(), dte);*

*} catch (ServiceFonctionnelleException sfe) {*

*LOGGER.info(sfe.getMessage());*

*throw sfe;*

*} catch (ServiceTechniqueException ste) {*

*LOGGER.error(ste.getMessage());*

*throw ste;*

*} catch (Exception e) {*

*LOGGER.error(e.getMessage(), e);*

*throw new ServiceTechniqueException(e.getMessage(), e);*

*}*

Le front peut ensuite rattraper les exceptions de type *Service,* et les gérer de façon différente selon le type *ServiceTechniqueException* ou *ServiceFonctionnelleException* et le code erreur.

### Logger

Un logger propre à l’ENT a été défini dans le core afin de s’abstraire de l’outil de log utilisé. Aucune dépendance externe ne sera alors explicitement importée dans le code source des applications. De plus, Slf4j, qui est utilisé pour les logs, étant volontairement demandé par le client, il ne fallait pas que la dépendance soit en dur dans le cas d’un changement. Toutes les classes des applications ENT doivent donc utiliser ce *Logger* customisé qui se trouve dans *org.lilie.socle.core.utils.log*.

## Mapping POJO

Les POJOs devant être transformés en DTOs afin de pouvoir transiter dans les couches supérieures, un mapping est assuré grâce à l’outil Dozer. Celui-ci est encapsulé dans le core par la classe *org.lilie.socle.core.utils.adapter.AdapterDozerImpl*, qui propose notamment des méthodes *mapToList()* pour gérer directement des listes et tableaux en plus d’un objet simple.

Des *Converter* spécifiques ont également été customisé pour l’ENT. Ils servent à convertir automatiquement les types simples sans que l’on ait besoin de les appeler explicitement dans les mappings. Ils sont simplement déclarés dans la configuration au début de chaque fichier de mapping. *ConverterBooleanInteger* permet de convertir automatiquement des 0/1 venant de la base de données en booléens Java, et *ConverterBooleanString* permet de convertir des O/N venant de la base de données en booléens Java.

## Conteneur Spring

Toutes les classes d’une application n’étant pas forcément chargées par Spring, certaines peuvent ne pas avoir accès directement au contexte. Afin de permette à n’importe qu’elle classe de récupérer le contexte Spring qui a été chargé et d’en utiliser les beans, le core propose la classe *org.lilie.socle.core.utils.spring.ApplicationContextHolder* qui met à disposition une méthode *static getContext()* permettant de récupérer le contexte.

## Gestion des fichiers de configuration des projets

### Objectif

Cette gestion concerne deux fichiers : le fichier de configuration des traces *log4j.xml*, et le fichier de configuration des paramètres de l’application *config.properties*. Ces deux fichiers doivent être externalisés du war/jar final afin d’être modifés facilement pour chaque environnement.

### Utilisation pour un module

* Dans le fichier Spring applicationContext.xml de la webapp :

*<bean id="propertyConfigurer" class="org.lilie.socle.core.utils.spring.CustomPropertyPlaceholderConfigurer">*

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

*<property name="fichierConfigVarEnvirStr" value="web<module>.config"/>*

*<property name="fichierConfigInterneStr" value="config.properties"/>*

*<property name="fichierLog4jVarEnvirStr" value="web<module>.log4j"/>*

*<property name="fichierLog4jInterneStr" value="log4j.xml"/>*

*<property name="delaiRechargementLog4jStr" value="60000"/>*

*</bean>*

* + Ainsi, la classe *CustomPropertyPlaceholderConfigurer* du core s’occupera du chargement des configs au lancement de l’application.
  + *configAppli* est un bean du contexte du core utilisé pour charger.
  + *fichierConfigInterneStr* et *fichierLog4jInterneStr* sont les chemins des fichiers en interne (utilisés par exemple pour l’environnement de développement lorsque les fichiers ne sont pas externalisés).
  + *fichierConfigVarEnvirStr* et *fichierLog4jVarEnvirStr* sont les noms des variables d’environnement contenant les chemins des fichiers externalisés.
  + Le délai de rechargement du log4j peut être modifié.
* Pour accéder à un paramètre du *config.properties* dans le code, il faut ensuite utiliser la classe du core *ConfigAppli* :
  + Définir dans votre classe un attribut « *ConfigAppli configAppli* », avec son setter, qui sera chargé par Spring (ne pas oublier de le mettre dans le bean xml de votre classe).
  + Utiliser par exemple la méthode *getProprieteStr(cle)* de *ConfigAppli*. Pour récupérer un paramètre du config d’une api, utiliser *getProprieteStr(cle, nomApi)*, le nom de l’api étant une constante Java présente dans chaque api.
* Pour externaliser les fichiers, voici comment modifier le packaging du projet :
  + Avoir un fichier *log4j.xml* et un fichier *config.properties* spécifiques au projet, placés à la racine de *src/main/resources.*
  + Ces fichiers doivent être conservés dans le war lorsque l’on build pour le développement, mais ils doivent être exclus du war et livrés séparément lors des packaging cibles. Pour cela, il faut modifier la configuration Maven de la manière suivante :
    - Dans le pom du projet, après la balise *<build>* à la racine de la balise *<project>* :

*<profiles>*

*<profile>*

*<id>packaging-cible</id>*

*<!-- Pour les environnements cibles, on ne met pas les fichiers de configuration dans*

*le war mais en externe avec des variables d'environnement -->*

*<build>*

*<resources>*

*<resource>*

*<directory>src/main/resources</directory>*

*<excludes>*

*<exclude>config.properties</exclude>*

*<exclude>log4j.xml</exclude>*

*</excludes>*

*</resource>*

*</resources>*

*</build>*

*</profile>*

*</profiles>*

* + - Dans le settings.xml, le Maven Profile *packaging-cible* ne doit pas être activé sur les postes de développement, mais doit l’être sur le poste de PIC qui fait le packaging de livraison :

*<activeProfiles>*

*<activeProfile>packaging-cible</activeProfile>*

*</activeProfiles>*

* Sur les environnements cibles, les fichiers sont récupérés grâce à des variables d’environnement. En développement, ces variables n’étant pas définies, par défaut les fichiers seront cherchés dans le war selon les chemins internes donnés dans la configuration Spring ci-dessus. Les noms des variables d’environnement à définir sont ceux utilisés dans la configuration Spring :

*web<module>.config*

*web<module>.log4j*

Y mettre les chemins vers les fichiers externalisés.

* Les fichiers étant externalisés, ils ne seront par défaut pas déployés avec le reste de la version lors d’une release. Afin qu’ils le soient, il est préférable d’ajouter un zip *assembly* dans la configuration Maven lors du build, qui sera déployé en même temps que le war avec le même numéro de version.

Dans le *profile* créé plus tôt, on ajoute donc la partie suivante :

*<profiles>*

*<profile>*

*<id>packaging-cible</id>*

*<!-- Pour les environnements cibles, on ne met pas les fichiers de configuration dans le jar*

*mais en externe avec des variables d'environnement -->*

*<build>*

*<resources>*

*<resource>*

*<directory>src/main/resources</directory>*

*<excludes>*

*<exclude>config\*.properties</exclude>*

*</excludes>*

*</resource>*

*</resources>*

***<!-- on crée donc un assemby zip pour les déployer à part -->***

***<plugins>***

***<plugin>***

***<groupId>org.apache.maven.plugins</groupId>***

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

***<configuration>***

***<descriptors>***

***<descriptor>src/main/assembly/config-descriptor.xml</descriptor>***

***</descriptors>***

***</configuration>***

***<executions>***

***<execution>***

***<id>config-assembly</id>***

***<phase>package</phase>***

***<goals>***

***<goal>attached</goal>***

***</goals>***

***</execution>***

***</executions>***

***</plugin>***

***</plugins>***

*</build>*

*</profile>*

*</profiles>*

Il faut bien sûr également ajouter le fichier *src/main/assembly/config-descriptor.xml* qui précise quoi mettre dans le zip :

*<assembly>*

*<id>config</id>*

*<formats>*

*<format>zip</format>*

*</formats>*

*<fileSets>*

*<fileSet>*

*<directory>${basedir}/src/main/resources</directory>*

*<outputDirectory>.</outputDirectory>*

*<includes>*

*<include>config\*.properties</include>*

*<include>log4j.xml</include>*

*</includes>*

*</fileSet>*

*</fileSets>*

*</assembly>*

### Fonctionnement interne

*CustomPropertyPlaceholderConfigurer*

Elle étend la classe *PropertyPlaceholderConfigurer* de Spring afin de charger en même temps le *log4j.xml* et le *config.properties*, mais également de centraliser la récupération des paramètres de l’application avec ceux des différentes apis utilisées. Elle permet notamment d’initialiser l’instance singleton du *ConfigAppli*.

Stockage des propriétés

Le bean *configAppli* contient à la fois les propriétés de l’application (dans un attribut *Properties*) et celles des apis utilisées par l’application (dans une *Map* de *Properties* dont la clé est la constante du nom de l’api). Le chargement de l’un ou l’autre des fichiers se fait de la même manière. La configuration Spring pour une api est juste légèrement différente afin de préciser la clé de stockage dans la map. Les apis n’ont également pas de *log4j.xml* spécifique car on utilise celui de l’application finale.

*<bean id="propertyConfigurerAnnuaire"*

*class="org.lilie.socle.core.utils.spring.CustomPropertyPlaceholderConfigurer">*

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

*<property name="fichierConfigVarEnvirStr" value="annuaire.config"/>*

*<property name="fichierConfigInterneStr" value="config\_api-annuaire.properties"/>*

***<property name="placeholderPrefix" value="$annuaire{"/>***

***<property name="nomApi" value="api-annuaire"/>***

*</bean>*

Le *placeholderPrefix* permet de récupérer les propriétés du bon *Properties* lors de l’utilisation au sein même du *applicationContext.xml* (cf exemple dans les *persistance-applicationContext.xml des apis*).

Initialisation du *ConfigAppli*

La méthode *charger()* s’occupe simplement du load des propriétés dans les bons attributs en fonction de si c’est pour l’application ou une api. La seule spécificité est l’utilisation de l’énumération *HiddenConfigEnum* qui permet de ne pas afficher les valeurs confidentielles dans les logs.

La méthode *init()* commence par charger le fichier de configuration des traces, puis récupère le flux du fichier de paramétrage pour le donner à la méthode *charger().*

La récupération du bon flux pour les fichiers est faite par l’intermédiaire de la classe *GestionConfig*. Que ce soit pour le *log4j.xml* (*chargerConfigLog4j()*) ou le *config.properties* (*getFluxConfigACharger()*), le traitement commence par regarder s’il existe une variable d’environnement afin de récupérer le fichier externalisé, puis s’il n’en trouve pas, il utilise le chemin interne.

*GestionConfig* contient également une méthode *getMsgSansLog4j()* qui permet de définir un format temporaire pour les logs à tracer avant même que le fichier de configuration des traces soit chargé.

## Gestion du cache

### Quand utiliser du cache

Le cache peut être géré à différents niveaux de l’application (infos en session, cache métier, cache du framework de persistance…). Nous nous intéressons dans ce paragraphe au cache métier.

Il ne sert à rien de mettre du cache dans chaque méthode business d’une application. C’est l’analyse au moment de la conception ou bien les tests de performances qui en détermineront le besoin ou non. Dans l’ENT, le cache est notamment pour l‘instant utilisé dans des cas où :

* des traitements sont répétés plusieurs fois pour des raisons techniques (par exemple la gestion de la sécurité et des droits d’accès aux urls)
* des traitements nécessitent l’appel à des serveurs distants que l’on ne maîtrise pas, afin d’éviter un bloquage éventuel (par exemple lecture de flux RSS, appels web services pour la web-conference…)

### Comment l’utiliser

Le cache métier de l’ENT est géré par l’intermédiaire de la librairie *ehcache*. Celle-ci permet entre autres de gérer du cache directement sur des méthodes existantes.

* La configuration se fait grâce au fichier *ehcache.xml* présent dans fmk-core-ent. Il suffit d’ajouter une balise *<cache>* avec un nom spécifique et les valeurs de paramètres voulues, à adapter selon les tests de performance.
* Ensuite, le traitement à mettre en cache se définit par annotation sur la méthode métier voulue :

*@Cacheable(cacheName = "<nomDuCacheDansXml>")*

A chaque appel de la méthode annotée, *ehcache* vérifie en cache s’il existe déjà la donnée de retour pour les mêmes valeurs d’entrée données en paramètres de la méthode.

* Si on ne précise rien, tous les paramètres de la méthode sont pris pour former la clé du cache. Si l’un des paramètres ne doit pas servir pour former la clé disciminante, il suffit de préciser les différents éléments de la clé avec l’annotation *@PartialCacheKey* devant chaque paramètre.

Dans l’exemple suivant, quelque soit la liste de groupes, le résultat renvoyé par le cache sera le même :

*@Cacheable(cacheName = "cacheDroits")*

*public GestionDroitsDto getGestionDroitsDto(@PartialCacheKey String typeUtilisateurStr,*

*@PartialCacheKey Long utilisateurId,*

*List < Long > groupeIdList,*

*@PartialCacheKey Long entiteId,*

*@PartialCacheKey boolean pagePubliqueBln)*

Attention :

Le cache étant géré par des aspects, il faut faire attention aux conflits lorsqu’il y a plusieurs annotations par aspects sur une même classe.

## Gestion de l’utilisateur connecté

Les informations de l’utilisateur sont mises en session dans chaque module par CAS lors de la connexion. Un *UtilisateurDto* permet de transporter ces données dans les différentes couches de l’application. Pour le type d’utilisateur, le core propose des méthodes dans *UtilisateurUtils* afin de déterminer par exemple si c’est un administrateur ou non, en fonction des valeurs des constantes communes. La récupération des informations au bon endroit en session est centralisée pour sa part dans fmk-core-web qui s’occupe de la partie front pour les webapps (cf chapitre suivant).

## Echanges de données entre modules

### Communication possible entre modules

Les modules sont sensés être le plus indépendants possible les uns des autres afin de respecter la modularité de l’ENT, aussi bien au niveau du code que du déploiement selon les différents porteurs de projet. Cependant, certains besoins ponctuels nécessitent un passage d’informations de l’un à l’eutre. Chaque module continue de gérer les entités métier et règles de gestion qui lui sont propres, mais peut mettre à disposition certaines données pour les autres modules.

Afin de garder une homogénéïté pour les flux de l’ENT, ces échanges doivent être faits avec la technologie REST. Un module met à disposition un service REST pour exposer ses propres entités, et ce service peut être appelé par les autres modules. L’api REST utilisée par l’ENT est Jersey.

### Contrat entre les deux modules

Le module appelant et appelé ne partagent pas de dto commun. Le « contrat » établi est sur la seule base d’une url http et d’un format xml de réponse.

Les constantes utilisées pour les échanges REST sont dans la classe *ContantesEchanges*. Ces constantes représentent les noms des paramètres et des balises xml échangées, définies par le « contrat » entre les deux modules.

Des constantes pouvant être utiles à plusieurs services sont déjà présentes, comme les noms des paramètres pour les informations de l’utilisateur connecté.

Le format du xml échangé doit respecter les nomenclatures suivantes :

* Format standard avec balise racine *<feed>* et une balise *<entry>* pour chaque ressource.
* Eviter d’ajouter pleins d’attributs spécifiques à la balise *<entry>* pour la garder le plus générique possible. Préférer les balises filles.
* Attribut « *service* » sur la balise *entry* pour y préciser le nom du module qui envoie la ressource.

### Utilisation par un module appelant

Afin de factoriser au maximum les traitements et de s’abstraire de l’outil Jersey, le core propose une api de gestion de ces échanges en encapsulant le fonctionnement du protocole REST, par l’intermédiaire du *EchangesBusinessImpl*.

Ce business propose une méthode qui encapsule l’appel REST lors d’un appel d’un module vers un autre :

*public String echangeDonneesEntreServices(NomAppliEnum service,*

*String urlRessourceStr,*

*Map < String, Object > paramsMap)*

On precise alors le module à appeler, la resource REST voulue et les paramètres associés.

La chaîne retournée est au format xml et doit donc être parsée par le module client afin d’en utiliser les données.

Dépendances

Les dépendances à ajouter dans le *pom.xml* de ce module sont :

*<dependency>*

*<groupId>jdom</groupId>*

*<artifactId>jdom</artifactId>*

*</dependency>*

*<dependency>*

*<groupId>com.sun.jersey</groupId>*

*<artifactId>jersey-client</artifactId>*

*</dependency>*

Les versions des librairies sont définies dans le pom parent ent.

### Mise à disposition de ressources par un module

Un module peut mettre à disposition une ou plusieurs ressources qui lui sont propres en exposant un service REST au lancement de la webapp. Le module garde la main sur le corps du traitement effectué par le service et donc sur les règles de gestion associées.

Url de la ressource

Il faut définir une url d’accès et des paramètres éventuels pour la ressource à exposer. Pour les ENT, un format commun a été défini. Ce pattern doit être respecté pour des raisons d’homogénéïté mais également de configuration du filtre CAS (les échanges REST sont des flux internes côté serveurs applicatifs ENT). Le pattern à utiliser pour l’url est précisé dans la constante *URL\_PATTERN\_COMMUN\_ECHANGES*. Le format final comprend donc cette constante suivie du pattern spécifique à la ressource demandée.

Exemple pour le module *Actualité* :

*@Path("/" + ConstantesEchanges.URL\_PATTERN\_COMMUN\_ECHANGES +*

*ConstantesEchanges.URL\_RESSOURCE\_ACTUALITE )*

Traitements du service

Après avoir choisi l’url de la ressource, il faut créer le service. Pour cela :

* Dans le package *business.echanges*, créer une classe correspondant à ce type de ressource. Pour la nomenclature, reprendre le pattern *EchangesRessource* avec le terme spécifique à la ressource devant. Par exemple pour Actualité : *EntiteEchangesRessource*.
* Ajouter la correspondance avec l’url en annotation sur la classe :

*@Path("/" + ConstantesEchanges.URL\_PATTERN\_COMMUN\_ECHANGES +*

*ConstantesEchanges.URL\_RESSOURCE\_ACTUALITE )*

*public class EntiteEchangesRessource {…}*

* Créer une méthode pour chaque type d’action possible sur la ressource (correspondant chacune à un code HTTP existant). Pour créer une méthode de consultation de la ressource avec un retour en plaintext, on ajoute les annotations suivantes à la méthode :

*@GET*

*@Produces("text/plain")*

* Si la méthode utilise des paramètres envoyés dans la requête HTTP, on les fait correspondre aux paramètres d’entrée de la méthode grâce à l’annotation *@QueryParam*.

Exemple :

*@GET*

*@Produces("text/plain")*

*public String getActualitesListXml*

*(@QueryParam(ConstantesEchanges.ECHANGES\_PARAM\_UTILISATEUR) Long utilisateurId,*

*@QueryParam(ConstantesEchanges.ECHANGES\_PARAM\_PORTEUR) String codePorteur) {…}*

L’annotation *@HeaderParam* fonctionne sur le même principe et peut être utilisée pour récupérer les informations d’authentification :

*@HeaderParam(ContainerRequest.AUTHORIZATION) String authentication*

* Le traitement de la méthode doit :
  + Vérifier que l’authentification est valide : utiliser la méthode *isAuthentificationOk()* de *EchangesBusiness* du core.
  + Appeler les traitements métiers propres à cette ressource
  + Transformer les résultats de cet appel en chaîne de caractères au format xml. (utilisation de la librairie *jdom* dans l’ensemble du code ENT)

Déclaration du service

* Dans le package *business.echanges*, créer une classe *EchangesApplication* qui étend *javax.ws.rs.core.Application*. Cette classe permet de déclarer les différentes ressources disponibles. On y met donc la déclaration de la classe *<nom>EchangesRessource* dans la méthode *getClasses()* prévue à cet effet.

*public Set < Class <?> > getClasses() {*

*Set < Class <?> > s = new HashSet < Class <?> >();*

*s.add(EntiteEchangesRessource.class);*

*return s;*

*}*

* Il faut ensuite indiquer cette classe à la webapp pour que les services soient disponibles au démarrage.

Dans le *web.xml*, ajouter :

*<!-- API REST -->*

*<filter>*

*<filter-name>Jersey Web Application</filter-name>*

*<filter-class>com.sun.jersey.spi.spring.container.servlet.SpringServlet</filter-class>*

*<init-param>*

*<param-name>javax.ws.rs.Application</param-name>*

*<param-value>org.lilie.services.actualite.business.echange.EchangesApplication</param-value>*

*</init-param>*

*</filter>*

*<filter-mapping>*

*<filter-name>Jersey Web Application</filter-name>*

*<url-pattern>/echanges/\*</url-pattern>*

*</filter-mapping>*

Dépendances

Les dépendances à ajouter dans le *pom.xml* de ce module sont :

*<dependency>*

*<groupId>jdom</groupId>*

*<artifactId>jdom</artifactId>*

*</dependency>*

*<dependency>*

*<groupId>com.sun.jersey</groupId>*

*<artifactId>jersey-server</artifactId>*

*</dependency>*

*<dependency>*

*<groupId>com.sun.jersey.contribs</groupId>*

*<artifactId>jersey-spring</artifactId>*

*</dependency>*

*<!-- dépendance sur le client aussi car EchangeBusinessImpl a l'import -->*

*<dependency>*

*<groupId>com.sun.jersey</groupId>*

*<artifactId>jersey-client</artifactId>*

*</dependency>*

Les versions des librairies sont définies dans le pom parent ent.

### Cas particulier des ressources de notifications

Plusieurs modules proposent de mettre à disposition des ressources de notifications permettant d’afficher les nouveautés à l’utilisateur sur sa page d’accueil. Afin que le traitement de récupération de ces ressources pour chaque module soit générique et centralisé, un format xml et des méthodes de traitements spécifiques ont été définies dans le projet fmk-core-ent. Ils sont à utiliser dans tout module voulant envoyer des notifications.

Le nom de la classe et l’url de la ressource sont les mêmes dans chaque module:

*@Path("/" + ConstantesEchanges.URL\_PATTERN\_COMMUN\_ECHANGES +*

*ConstantesEchanges.URL\_COMMUNE\_RESSOURCE\_NOTIFICATIONS)*

*public class NotificationsEchangesRessource {…}*

Les constantes spécifiques aux notifications, pour le format du xml et les paramètres de l’appel, sont présentes dans le fichier *ConstantesEchanges.*

Deux méthodes sont disponibles dans *EchangesBusiness* :

*public String genereXmlNotifications(List < NotificationDto > notificationDtoList)*

*public List < NotificationDto > parseXmlNotifications(String notificationsReponseXml)*

Elles doivent être utilisées :

* Par le module qui propose le service, pour transformer les dto de notifications en flux xml
* Par le module appelant, pour parser le flux xml reçu et le transformer en dto utilisables par les autres couches

Ce sont les dto *NotificationDto* et *NotificationContentDto* qui sont utilisés. Un *NotificationDto* est composé de plusieurs contenus ayant chacun leur paramétrage pour mettre en valeur une partie du texte ou un lien.

### Fonctionnement interne

Traitements internes methodes core, Le format du xml notif à preciser

Dans le cas d’un besoin de modification des traitements centralisés dans fmk-core-ent, en voici le fonctionnement interne.

*echangeDonneesEntreServices()*

L’appel REST se base sur l’api Jersey. On crée donc un client Jersey auquel on passe toutes les informations nécessaires pour déterminer la ressource (url, paramètres).

Le mode d’authentification utilisé pour les échanges REST entre modules est le Basic HTTP. Ce ne sont que des flux internes aux applications ENT qui ne sont pas en HTTPS. L’ajout des informations d’authentification se fait par un filtre précisé sur le client Jersey.

L’url du contexte, le login et le mot de passe sont différents pour chaque module et sont à configurer dans le fichier *config\_fmk-core-ent.properties*. L’énumération *ContexteAppliConfigEnum* propose une méthode pour récupérer les noms des propriétés dans le fichier pour chaque service. A partir de ces noms, l’objet *ConfigAppli* peut être utilisé pour récupérer les valeurs des paramètres, comme décrit dans les chapitres précédents.

L’objet réponse de Jersey permet de récupérer le statut de l’appel en plus du résultat. On vérifie donc le statut *OK 200* avant de parser la réponse xml.

*isAuthentificationOk()*

Le module appelé vérifie les points suivants :

* Le préfixe d’authentification doit correspondre au Basic HTTP.
* Les valeurs ne doivent pas être nulles et bien formattées.
* La chaîne envoyée est décodée (Base64) et le login/mdp est comparé aux valeurs du *config\_fmk-core-ent.properties* comme décrit juste au-dessus.

*genereXmlNotifications()*

A partir d’une liste de dto spéciaux pour les notifications, le traitement utilise JDOM pour générer une chaîne de caractères au format xml.

Le format des notifications respecte les règles générales précisées ci-dessus pour l’ensemble des échanges.

*<feed>*

*<entry service="blog">*

*<content emphasize="false" url="urlVersLeBlog">Le blog</content>*

*<content emphasize="true" url="urlVersLeBlog">théâtre</content>*

*<content emphasize="false" url="">a été ajouté.</content>*

*</entry>*

*<entry service="blog">*

*<content emphasize="false" url=" ">Le blog</content>*

*<content emphasize="true" url="">mathématiques</content>*

*<content emphasize="false" url="">a été supprimé.</content>*

*</entry>*

*</feed>*

*parseXmlNotifications()*

Cette méthode fait l’exact traitement inverse, également à l’aide JDOM.

## Gestion des fichiers uploadés

### Objectif

Plusieurs modules ont besoin de stocker des fichiers et images sur le serveur applicatif (pièces jointes de messages, photos de profil, images de blogs…). Afin d’abstraire les modules de la connaissance du stockage réel et de centraliser les traitements communs, le core propose une classe d’encapsulation pour ces besoins. Ceci permet au socle de maîtriser l’ensemble de la gestion des fichiers et d’avoir éventuellement la main sur l’espace de stockage utilisé.

Le business à utiliser pour gérer des fichiers est donc *FichiersBusinessImpl*.

### Sécurisation

Egalement, cette centralisation permet au core de crypter les urls des fichiers afin que le chemin n’apparaisse jamais en clair côté client. Le navigateur n’obtient jamais l’url décryptée. Il est alors nécessaire de repasser par le core avec la bonne gestion des droits pour récupérer un fichier.

### Discriminant d’un fichier

Dans le cadre de cette encapsulation, un fichier n’est donc pas discriminé par son chemin (dont la totalité n’est connue que du core) mais par ses identifiants métier :

* Nom du fichier
* Id de l’entité à laquelle il est associé (l’id du blog ou de la personne par exemple)

L’objet *InfosFichierDto* permet de regrouper ces informations. Le chemin de stockage est ensuite déterminé par le *FichiersBusiness* en fonction de ces informations.

### Upload d’un fichier

Afin d’uploader un fichier, il suffit de fournir les informations discriminantes et le fichier lui-même à la méthode suivante :

*public void uploadFichier(Long idDonnee, String nomFichier, File fichier)*

### Récupération d’un fichier stocké

Pour consulter un fichier stocké, la seule chose à faire par le module est d’appeler la méthode suivante du *FichiersBusiness* :

*public String getUrlFichier(Long idDonnee, String nomFichier)*

et de mettre le résultat de cet appel dans la source HTML.

Une méthode de suppression d’un fichier est également présente avec les mêmes paramètres.

### Fonctionnement interne

*Récupération d’un fichier*

On distingue deux étapes :

* Récupération de l’url à mettre dans la source de la page HTML renvoyée au navigateur : Cette url est cryptée. On la récupère grâce à la méthode *getUrlFichier().*
* Lors du clic sur le fichier dans la page, celui-ci ne peut pas ouvrir directement le fichier. Il faut repasser par le serveur pour le décryptage. On utilise alors la méthode :

*public File getFichier(String cheminCrypteStr)*

Afin de ne pas avoir à refaire ce travail de passage par une action du serveur différemment dans chaque module, le projet *fmk-core-web* propose une encaspulation pour la partie frontale.

En réalité, l’url renvoyée par *getUrlFichier()* n’est pas seulement le chemin du fichier, mais l’url d’une action Struts du *fmk-core-web*, qui se charge d’effectuer le traitement d’appel du *FichiersBusiness* pour le décryptage.

Le chemin crypté est fourni dans cette url en paramètre de la requête à l’action. L’appel à *getFichier(String cheminCrypteStr)* est donc fait par cette action, que nous détaillerons dans les chapitres suivants concernant *fmk-core-web*.

*Construction de l’url de stockage*

La méthode

*public String getCheminFichier(Long idDonnee, String nomFichier)*

est utilisée en interne du business afin de construire le chemin complet dans l’espace de stockage, à partir des informations métier fournies. Le chemin absolu est récupéré dans les fichiers de configuration des applications.

Cette méthode est utilisée à la fois par la méthode d’upload et celles de consultation.

*Cryptage de l’url*

Le cryptage/décryptage du chemin est fait à l’aide de la classe *CryptageUtils* décrite dans les chapitres suivants.

## Encapsulation d’api FTP

### Description générale

Le core propose également un business pour gérer des accès distants avec le protocole FTP. On peut utiliser soit le protocole FTP, soit sa variante sécurisée FTPS.

Le *FtpBusiness* s’utilise en singleton comme tout autre business du core. Il propose divers méthodes permettant la consultation et modification à distance des fichiers sur un serveur FTP déterminé.

Les informations de connexion doivent être repassées systématiquement aux diverses méthodes (par l’intermédiaire du *ConnexionFtpDto*), afin que le business encapsule toute la partie connexion et reconnexion éventuelle lors d’une perte de session. Il n’y a donc pas de méthode de connexion à proprement parlé qui puisse être appelée en public, mais bien une méthode de déconnexion.

### Fonctionnement interne

L’implémentation du business est basée sur l’api *commons-net* qui est encapsulée et dont aucun objet n’apparaît donc dans la signature des méthodes.

L’objet de référence utilisé dans le business pour chaque connexion FTP est le *org.apache.commons.net.ftp.FTPClient*. Le *FTPSClient* utilisé pour le protocole FTPS hérite de la classe *FTPClient*, donc à part au moment de le créer, on utilise l’un ou l’autre indifféremment sous le type *FTPClient* commun. Les traitements appliqués sont identiques.

Le business fonctionne avec une map contenant les différentes connexions en cours. A chaque fois qu’un appel est fait au business, il vérifie s’il existe déjà une connexion valide pour les paramètres du *ConnexionFtpDto* fourni. Il réutilise la connexion existante ou en crée une nouvelle si besoin. Le code appelant se retrouve donc toujours avec une connexion valide sans s’en soucier.

Les paramètres url et login seuls ne sont pas forcément discriminants pour déterminer une session de connexion unique. Plusieurs connexions peuvent éventuellement être demandées sur le même login (possibilité d’un login commun à tous les enseignants ou un groupe…). Afin d’avoir une clé discriminante, le business génère un identifiant unique d’après une séquence conservée dans la classe. Cet identifiant sert de clé dans la map et est ensuite fourni au code appelant par l’intermédiaire de la référence sur le *ConnexionFtpDto*. Lorsque le code appelant redonne donc plus tard son *ConnexionFtpDto*, celui-ci contient l’identifiant (attribut *clientId*) servant à retrouver l’objet de connexion dans la map.

## Autres utilitaires divers

### Outil de cryptage

La classe CryptageUtils permet d’encapsuler les traitements d’encryption nécessaires aux différents modules.

TODO à décrire après refonte due au ftp.

### Email

La classe *EmailUtils* propose des méthodes d’envoi d’e-mail par SMTP. Elle se base sur l’api *javax.mail*.

Le serveur à utiliser est à configrer dans le fichier de configuration du core.

### Libellés

Certains modules n’utilisent pas Struts ou un autre framework encapsulant la gestion des libellés d’interface (par exemple pour une applet). Afin d’avoir tout de même une gestion des libellés de manière externalisée et internationalisable, le core propose une encapsulation basique :

La classe *org.lilie.socle.core.utils.constantes.LibelleHolder* permet de gérer le chargement et la consultation des libellés. Le fichier de libellés est à mettre à la racine de *src/main/resources* sour le nom *libelles.properties*.

### Constantes

Le projet *fmk-core-ent* permet également de centraliser les constantes et dtos qui peuvent être communs aux différents modules, comme par exemple les types de profil, les noms des modules (*Constantes.java*), l’objet utilisateur…

## Configuration du core

La configuration se trouve dans *src/main/resources*.

Le fichier de configuration du core est le fichier *config\_fmk-core-ent.properties*. Il est chargé de a même manière que les autres fichiers de configuration des apis, comme décrit plus haut. Toute propriété paramétrable du projet doit être dans ce fichier afin d’être externalisée.

Pour l’instant, il contient les informations d’authentification pour les échanges entre modules, ainsi que le serveur SMTP à utiliser pour l’envoi de mail.

Les libellés utilisés par le projet fmk-core-ent doivent également être externalisés du code pour des raisons de maintenance et d’internationalisation. Ils sont à mettre dans le fichier *messages\_fmk-core-ent.properties*.

## Packaging du projet

Le projet fmk-core-ent doit être utilisé par tous les projets ENT, en le déclarant de la même manière qu’une librairie externe.

Le projet fmk-core-ent n’a pas de dépendance interne ENT car c’est le projet de base dont tous les autres se servent.

Les dépendances vers les librairies externes utilisées sont quant à elles déclarées en optionnel de manière à ne pas les embarquer à chaque build. Seules les libraires réellement utilisées par l’application appelante seront packagées. L’application appelante doit donc repréciser, dans son propre *pom.xml*, les dépendances dont elle a besoin selon les fonctionnalités du core utilisées.

# Fmk-core-web

## Présentation

Le projet fmk-core-web propose le même type de service que le fmk-core-ent mais pour la partie frontale des webapps. On y trouve des utilitaires pour la vue (ressources web classiques + JSP) et le contrôleur (Struts 2). Toutes les webapps sur le modèle de front en Struts 2 dépendent donc de ce projet.

Certaines fonctionnalités décrites pour le projet fmk-core-ent trouvent leur suite ici pour la partie front.

## Classes de référence

La principale classe de référence proposée par le projet est la classe mère pour la couche contrôleur. Il s’agit de la classe *Action*, héritant de la classe *ActionSupport* de Struts.

Toutes les actions des webapps en héritent afin de centraliser les besoins communs.

La classe propose pour l’instant deux attributs :

* choixTraitementStr : permet de préciser l’action de redirection à choisir.
* provenanceStr : permet de conserver une indication sur la page appelante pour pouvoir faire un retour éventuel, depuis une page accédée de plusieurs endroits.

## Gestion de l’utilisateur connecté

La classe org.lilie.socle.core.web.actions.UtilisateurUtils, héritant de celle de fmk-core-ent, permet de récupérer facilement les informations de l’utilisateur connecté données par CAS et le portail. Qu’elles soient en session ou dans le header de la request, le traitement est centralisé par cet utilitaire. Les méthodes permettent de récupérer les différents éléments de l’objet *UtilisateurDto* (identifiant, type, groupes, porteur).

## Gestion des données en session

### Objectif

Mettre des données en session pour réutilisation entre les différentes pages permet de faire le lien entre les différents éléments de navigation mais également de gagner en performance. Cependant, si l’on a tendance à mettre trop de choses en session, celle-ci peut devenir surchargée et provoquer l’effet inverse. Une maîtrise des données en session est donc nécessaire. Plusieurs solutions peuvent être envisagées pour purger ou autre… Dans le cas des ENT, nous avons opté pour une gestion de conversations au sein de la session.

### Utilisation des conversations

Une conversation est une sorte de tuyau de données partagé entre plusieurs actions Struts. Les données sont bien en session, mais « rangées » par conversation.

* Cela fonctionne sur un principe d’abonnement :
* On peut avoir autant de conversations que nécessaire (ni trop, ni trop peu, à mesurer selon le besoin fonctionnel et l’efficacité de la purge voulue).
* Une action, méthode ou classe, peut s’abonner à cette conversation par l’intermédiaire de l’annotation :

*@ConversationSupport({liste des conversations})*

Les noms des conversations sont à donner à partir des constantes à définir dans la classe *ConstantesConversation* de la webapp.

* Pour ajouter une donnée dans une conversation à laquelle on est abonné, il faut utiliser la ligne suivante :

*ConversationManager.put(constanteNomConversation,*

*constanteNomDonneeEnSession,*

*valeur);*

* Pour récupérer une donnée :

*ConversationManager.get(constanteNomConversation,*

*constanteNomDonneeEnSession);*

* Chaque conversation qui n’est plus utilisée est purgée de la session. Une conversation non utilisée signifie : lorsque l’on passe dans une action, chaque conversation à laquelle l’action n’est pas abonnée est considérée comme non utilisée.
* La conversation *ConversationConst.PRESERVE\_ALL* revient à s’abonner à toutes les conversations pour n’en purger aucune.

### Fonctionnement interne

Les classes gérant les conversations en session sont dans le package *org.lilie.socle.core.web.actions.conversation*.

Struts 2 propose un système d’intercepteurs au niveau des actions. Celui-ci nous permet de réaliser un traitement particulier au début de chaque action. L’ENT met donc à disposition :

* Un *ConversationInterceptor* qui gère ce qui est relatif à l’action:

La méthode *intercept()* utilise le *ConversationHandler* qui définit la méthode de purge en fonction des abonnements.

La méthode *getAllowedConversations()* permet de récupérer les abonnements de l’action en fonction des annotations sur la méthode ou sur la classe.

* Un *ConversationHandler* qui gère les conversations en elle-mêmes : purge, création d’une conversation, suppression d’une conversation.
* Un *ConversationManager* qui met à disposition des méthodes d’ajout et récupération de données dans les conversations. Il interagit avec le *Conversationhandler* et la session directement.

## Gestion des erreurs

### Gestion dans le contrôleur

Comme vu plus haut, les exceptions remontées jusqu’au contrôleur sont les exceptions du business, c'est-à-dire les *ServiceFonctionnelleException* et *ServiceTechniqueException*, ainsi que les exceptions de type générique Exception du corps du contôleur.

C’est le contrôleur qui doit ensuite choisir comment les traiter. Les exceptions fonctionnelles peuvent générer des messages à l’utilisateur en fonction du code erreur rattrapé, les exceptions techniques peuvent renvoyer vers la page d’erreur générique de l’ENT. Cette dernière est décrite ci-dessous.

Pour le cas particulier des traitements Ajax, les exceptions techniques ne doivent pas retourner vers le page d’erreur directement car le Javascript appelant ne sait pas comment le gérer proprement. Dans le cas d’une action utilisée pour un appel Ajax, la méthode ne doit pas forwarder vers « ERROR », mais utiliser plutôt la ligne de code suivante dans le catch :

*ExceptionUtils.retourneErreurAjax();*

### Pages d’erreur

Le projet fmk-core-ent met à disposition une action *AffichagePageErreurAction* qui dirige vers une JSP spécifique. Cette action est déclarée pour chaque webapp globalement comme étant appelée lors de l’utilisation du forward Struts « *ERROR* ».

La page d’erreur est définie dans *webapp/jsp/error/error.jsp*. Cette JSP utilise les taglibs spécifiques aux ENT *MessageError* et *StacktraceError*, qui permettent d’afficher notamment la stacktrace.

De manière similaire, les exceptions concernant les erreurs de droits (*AccesNonAutoriseException*) sont déclarées globalement dans Struts comme étant rattrapées par l’action *AffichageAccesNonAutoriseAction*, associée à *accesNonAutorise.jsp*.

### Page de supervision

Une servlet permettant la supervision est déclarée dans le *web.xml* de chaque module, afin de permettre d’afficher une page résumant le fontionnement ou non des différents services techniques utilisées (bases de données, serveur d’indexation, chargement du fichier de config...), et la version en cours.

Cette servlet est définie dans fmk-core-web : *MaintenanceServlet*. Chaque module a sa propre Servlet héritant de celle-ci afin de repréciser ses différents services spécifiques.

TODO reprendre la doc de MLAM

## Export

### Utilisation

La classe *ExportUtils* propose une méthode *ouvrirFichier()* qui permet d’ouvrir un fichier d’export directement dans le navigateur. Il suffit d’appeler cette méthode avec en paramètre le fichier à ouvrir, et de retourner *null* à la fin de l’action.

### Fonctionnement

Avec l’utilisation de Struts, afin d’ouvrir un fichier, il faut bypasser le flux normal de la requête HTTP. Au lieu de faire le forward Struts habituel, on redirige la réponse sur le contenu du fichier.

## Gestion des fichiers uploadés

### Contrôleur

Comme dit dans le chapitre sur les fichiers uploadés dans fmk-core-ent, il existe une action *RecuperationFichierAction* qui permet de gérer le décryptage de l’url et l’ouverture d’un fichier stocké sur le serveur. C’est le *path* Struts de cette action, déclarée dans le *struts.xml* de fmk-core-web, qui fait partie de l’url renvoyée par la méthode *getUrlFichier()* de *FichierBusiness*.

Le traitement de cette action est le suivant :

* Grâce au *FichierBusiness*, récupération du fichier à partir du chemin crypté passé en request
* Utilisation de ExportUtils pour ouvrir le fichier en question

### Utilitaires pour la vue

Les formulaires permettant l’upload et l’affichage / aperçu des images et des pièces jointes dans les applications sont semblables. Ces formulaires ont donc été centralisés dans fmk-core-web pour avoir des traitements homogènes dans toutes les applications.

Trois fragments de JSP ont été définis et peuvent être inclus dans les JSP de l’application nécessitant une gestion d’upload d’images et de pièces jointes (*<jsp:include page="/jsp/uploads/formulaire.jsp" />* par exemple) :

* */jsp/uploads/fomulaire.jsp* : ce fragment représente le formulaire d’upload suivant :



* */jsp/uploads/apercu.jsp* : ce fragment gère l’aperçu de l’entité (actualité, blog etc…) avec l’image positionnée et redimensionnée et l’affichage de la pièce jointe. Cet aperçu est effectué en javascript en fonction du choix effectué dans le formulaire ci-dessus
* */jsp/uploads/affichage.jsp* : ce fragment gère l’affichage à proprement parler (et non plus l’aperçu) de l’entité avec l’image positionnée et redimensionnée et l’affichage de la pièce jointe en fonction des informations qui ont été stockées en base au moment de l’enregistrement de l’entité.

NB : Les JSP *apercu.jsp* et *affichage.jsp* ne gèrent pas la mise en forme du contenu propre à l’entité (contenu de l’actualité, du blog etc…).

Pour les fragments *formulaire.jsp* et *apercu.jsp* (qui normalement sont inclus dans la même JSP), il faut définir et valoriser au sein de l’action un objet *UploadsDto* (DTO de fmk-core-ent) nommé ***uploadsDto***. Ce DTO regroupe toutes les propriétés utiles aux JSP : positionnement de l’image, fichier uploadé, redimensionnement, masquage ou non de la partie de formulaire de gestion des images, masquage ou non de la partie de formulaire de gestion des pièces jointes etc…

Cet objet doit être utilisé dans les actions pour passer les propriétés de l’action aux JSP et inversement.

Un javascript */js/upload.js* (qui doit être importé dans la JSP qui inclut *formulaires.jsp* et *affichage.jsp*) permet de positionner l’image en mode « aperçu ». La fonction à appeler pour ce faire est *apercuImage*. Cette fonction doit être appelée dans la fonction javascript qui bascule l’affichage en mode aperçu.

Exemple :

*function apercu(bool) {*

*if (bool) {*

*// on fait disparaitre la creation et apparaitre l'apercu*

*// on met en forme le contenu de l’entité et on l’affiche*

*...*

*$('#contenu').html(contenuFinal);*

*// apercu de l'image*

*apercuImage(bool, lageurMaximaleImage);*

*} else {*

*// on fait disparaitre l’apercu et apparaitre la creation*

*...*

*}*

*}*

### Fonctionnement interne

Lien entre les utilitaires de la vue et les méthodes java du core

Les données stockées en base ne sont en aucun cas les chemins sur le serveur de stockage, puisque l’objectif est justement de les crypter et d’en centraliser la gestion. On ne stocke donc que les données discriminantes de chaque fichier, c'est-à-dire les données permettant de récupérer le fichier ensuite (nom et id de la donnée comme dans *InfosFichierDto*).

Le code des JSP et JS ci-dessus contient les appels effectifs vers l’action de récupération décrite plus haut et les méthodes java du core.

Détails du fonctionnement

TODO LBIL

## Ressources statiques communes

### Objectif

Les ressources web statiques communes aux différentes webapps sont centralisées dans fmk-core-web. Cela permet de ne les modifier qu’à un seul endroit et de fournir un socle web. Elles sont toutes dans le dossier *webapp* du projet.

Pour l’utilisation depuis les webapps, cf le chapitre suivant sur le packaging.

### Ressources pour le look

Les images et feuilles de styles CSS de l’ENT sont toutes stockées dans ce projet, dans les dossiers *webapp/css* et *webapp/img*. Le look des modules doit être homogène et respecter la charte graphique. Il existe une documentation séparée expliquant l’utilisation des différents styles dans les pages.

### Utilitaires Javascript et JSP

Les fichiers javascripts sont également centralisés. Les webapps définissent des fichiers javascript spécifiques à leurs pages, mais les traitements communs sont à mettre dans fmk-core-web, dans l’arborescence fonctionnelle appropriée du dossier *webapp/js*.

Des utilitaires divers sont déjà présents, comme par exemple :

* des fonctionnalités de la librairie JQuery adaptées pour l’ENT : gestion des appels Ajax avec *ENT-ajax.js*, gestion des calendriers avec *ENT-calendar.js*, gestion des réglettes avec ENT-slider.js. (Les scripts encapsulant à ajouter à la page sont ajax.js, calendar.js et slider.js)
* gestion des fichiers à uploader (JS+JSP), détaillé dans les chapitres précédents en même temps que le code java associé
* éditeur de WYSIWYG : utilisation de l’outil TinyMCE
* jsp pour les pages d’erreur

### Autres

D’autres ressources sont également partagées comme les TLDs pour les taglibs des JSP.

## Configuration

Le projet fmk-core-web contenant des actions Struts, pour pouvoir les utiliser, il faut importer le *struts\_fmk-core-web.xml* dans le fichier *struts.xml* de la webapp :

*<include file="struts\_fmk-core-web.xml"/>*

Ce projet n’a pas de fichier *config.properties*.

## Packaging

### Dépendances

Le projet fmk-core-web doit être utilisé par tous les projets ENT de type webapp Struts 2, en le déclarant de la même manière qu’une librairie externe. Il n’est alors pas utile de redéclarer les dépendances et imports vers fmk-core-ent, car ils sont embarqués par fmk-core-web.

Le projet fmk-core-web a deux dépendances internes ENT :

* fmk-core-ent
* api-portail (et par transitivité les apis utilisées par api-portail)

Comme pour fmk-core-ent, les dépendances vers les librairies externes utilisées sont quant à elles déclarées en optionnel de manière à ne pas les embarquer à chaque build. Seules les libraires réellement utilisées par l’application appelante seront packagées. L’application appelante doit donc repréciser, dans son propre *pom.xml*, les dépendances dont elle a besoin selon les fonctionnalités du core-web utilisées.

### Déploiement des ressources web

Les ressources web centralisées dans fmk-core-web doivent être packagées de manière à être utilisables par les webapps.

Lors du build, un assembly Maven est donc généré en plus du jar. Cette tâche est associée dans le pom.xml au goal Maven Package.

*<plugin>*

*<groupId>org.apache.maven.plugins</groupId>*

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

*<configuration>*

*<descriptors>*

*<descriptor>src/main/assembly/web-resources-descriptor.xml</descriptor>*

*</descriptors>*

*</configuration>*

*<executions>*

*<execution>*

*<id>web-resources-assembly</id>*

*<phase>package</phase>*

*<goals>*

*<goal>attached</goal>*

*</goals>*

*</execution>*

*</executions>*

*</plugin>*

Le *descriptor* de l’*assembly* permet de définir les chemins des ressources à mettre dans le zip en question.

*<assembly>*

*<id>web-resources</id>*

*<formats>*

*<format>zip</format>*

*</formats>*

*<fileSets>*

*<fileSet>*

*<directory>${basedir}/src/main/webapp/img</directory>*

*<outputDirectory>/img</outputDirectory>*

*</fileSet>*

*<fileSet>*

*<directory>${basedir}/src/main/webapp/js</directory>*

*<outputDirectory>/js</outputDirectory>*

*</fileSet>*

*<fileSet>*

*<directory>${basedir}/src/main/webapp/jsp</directory>*

*<outputDirectory>/jsp</outputDirectory>*

*</fileSet>*

*<fileSet>*

*<directory>${basedir}/src/main/webapp/tld</directory>*

*<outputDirectory>/tld</outputDirectory>*

*</fileSet>*

*<fileSet>*

*<directory>${basedir}/src/main/webapp/css</directory>*

*<outputDirectory>/css</outputDirectory>*

*</fileSet>*

*</fileSets>*

*</assembly>*

Le zip est déployé sur le repository Nexus en même temps que le jar. Il reprend le nom et la version de l’artifact avec un *classifier* « *web-ressources* » en plus.

### Récupération des ressources web par les webapps

Les webapps ont accès au jar et au zip sur Nexus. En plus de la dépendance sur le fmk-core-web dans leur *pom.xml*, il faut déclarer une dépendance sur l’artifact avec le classifier « *web-ressources* ».

*<dependency>*

*<groupId>org.lilie.socle</groupId>*

*<artifactId>fmk-core-web</artifactId>*

*</dependency>*

*<dependency>*

*<groupId>org.lilie.socle</groupId>*

*<artifactId>fmk-core-web</artifactId>*

*<version>${pom.version}</version>*

*<classifier>web-resources</classifier>*

*<type>zip</type>*

*<scope>provided</scope>*

*</dependency>*

Ensuite, lors du build, deux plugins Maven sont configurés pour utiliser ces ressources :

* maven-dependency-plugin : afin de récupérer et dézipper les ressources web dans le répertoire de travail
* maven-ressources-plugin : afin de copier les ressources web dézippées au bon endroit dans le target de la webapp, pour qu’ils soient pris en compte dans la création du war final

*<plugin>*

*<groupId>org.apache.maven.plugins</groupId>*

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

*<executions>*

*<execution>*

*<id>execution\_fmk-core-web</id>*

*<configuration>*

*<outputDirectory>${assembly.work}</outputDirectory>*

*<includeArtifacIds>fmk-core-web</includeArtifacIds>*

*<includeGroupIds>org.lilie.socle</includeGroupIds>*

*<includeClassifiers>web-resources</includeClassifiers>*

*<excludeTransitive>true</excludeTransitive>*

*<excludeTypes>jar</excludeTypes>*

*<overWriteIfNewer>true</overWriteIfNewer>*

*<overWriteSnapshots>true</overWriteSnapshots>*

*<overWriteReleases>true</overWriteReleases>*

*</configuration>*

*<phase>generate-resources</phase>*

*<goals>*

*<goal>unpack-dependencies</goal>*

*</goals>*

*</execution>*

*</executions>*

*</plugin><plugin>*

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

*<executions>*

*<execution>*

*<id>copy-tld</id>*

*<phase>generate-resources</phase>*

*<goals>*

*<goal>copy-resources</goal>*

*</goals>*

*<configuration>*

*<outputDirectory>${dir.tld}</outputDirectory>*

*<resources>*

*<resource>*

*<directory>${dir.tld.assembly}</directory>*

*</resource>*

*</resources>*

*</configuration>*

*</execution>*

*<execution>*

*<id>copy-img</id>*

*<phase>generate-resources</phase>*

*……*

*</execution>*

*……*

*</executions>*

*</plugin>*