**Spécification technique – Guide de développement d’un module web**

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

**de la région Île-de-France**

**Spécification Technique   
Guide de développement d’un module web full Lilie**

**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 | 29/06/2011 | MMAU | Initialisation du document |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Sommaire

[Introduction 5](#_Toc297800880)

[1. Architecture applicative 5](#_Toc297800881)

[1.1. Architecture en couches du module 5](#_Toc297800882)

[1.1.1. Principe 5](#_Toc297800883)

[1.1.2. La couche « présentation » ou « vue » 6](#_Toc297800884)

[1.1.3. La couche « coordination » ou « contrôleur » 6](#_Toc297800885)

[1.1.4. La couche « service » 6](#_Toc297800886)

[1.1.5. La couche « domaine » ou « métier » 7](#_Toc297800887)

[1.1.6. La couche « persistance » 7](#_Toc297800888)

[1.1.7. Architecture en couches et approche Framework 7](#_Toc297800889)

[1.2. Architecture applicative globale 9](#_Toc297800890)

[2. Création du module 10](#_Toc297800891)

[3. Règles générales 10](#_Toc297800892)

[4. Développement de la couche contrôleur 11](#_Toc297800893)

[4.1. Présentation 11](#_Toc297800894)

[4.2. Classes de référence 12](#_Toc297800895)

[4.3. Création des classes actions 12](#_Toc297800896)

[4.3.1. Organisation fonctionnelle 12](#_Toc297800897)

[4.3.2. Nommage 12](#_Toc297800898)

[4.3.3. Implémentation 13](#_Toc297800899)

[4.3.4. Spring 13](#_Toc297800900)

[4.4. Navigation 13](#_Toc297800901)

[4.4.1. Fonctionnement Struts 13](#_Toc297800902)

[4.4.2. Init et Fin 14](#_Toc297800903)

[4.5. Gestion des exceptions 15](#_Toc297800904)

[4.6. Validation des données 16](#_Toc297800905)

[4.6.1. Principe 16](#_Toc297800906)

[4.6.2. Validator Struts 16](#_Toc297800907)

[4.7. Conversations 17](#_Toc297800908)

[4.8. Sécurité 17](#_Toc297800909)

[4.8.1. Objectif 17](#_Toc297800910)

[4.8.2. Solution choisie 18](#_Toc297800911)

[4.8.3. Utilisation par le module 18](#_Toc297800912)

[4.8.4. Fonctionnement interne 20](#_Toc297800913)

[5. Développement de la couche vue 21](#_Toc297800914)

[5.1. Découpage 21](#_Toc297800915)

[5.2. Javascript 21](#_Toc297800916)

[5.3. JSP 22](#_Toc297800917)

[5.4. Images et CSS 24](#_Toc297800918)

[5.5. Ressources du fmk-core-web 24](#_Toc297800919)

[5.6. Sitemesh 24](#_Toc297800920)

[5.6.1. Objectif 24](#_Toc297800921)

[5.6.2. Définition du template 25](#_Toc297800922)

[5.6.3. Utilisation pour les pages 26](#_Toc297800923)

[5.6.4. Configuration de la webapp 27](#_Toc297800924)

[5.7. Web.xml 28](#_Toc297800925)

[6. Constantes et utilitaires 29](#_Toc297800926)

[7. Packaging 29](#_Toc297800927)

[Annexe 30](#_Toc297800928)

# 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 l’architecture applicative générale d’un module web full Lilie, mais également le détail de la partie front-office web. La partie back-office et les règles de développement communes aux autres types de projets (batch, apis…) sont décrites dans le guide de développement général.

# Architecture applicative

## Architecture en couches du module

### Principe

L’orientation retenue pour l’architecture applicative est la suivante :

* Choix de la technologie J2EE (Java 2 Enterprise Edition) de Sun afin de garantir l’indépendance des applications développées par rapport aux plateformes technologiques et par rapport aux éditeurs.
* Utilisation de briques Open Source pertinentes et fiables répondant aux besoins spécifiques du projet. Ce choix permet de limiter les coûts de licence, de bénéficier de la dynamique Open Source, de limiter la dépendance par rapport aux éditeurs.
* Utilisation du conteneur léger Spring. Ce choix permet de proposer une base commune pour la structuration des différentes parties du code, tout en simplifiant les développements et les tests.

En pratique, la mise en place de ceci repose sur deux principes :

* une architecture applicative structurée en couches
* une approche Framework

L'architecture applicative de l’ENT est basée sur plusieurs couches logiques. Ceci permet la séparation des différentes logiques applicatives dans le but de faciliter la maintenance, l’extensibilité et la capacité de montée en charge de l’application.



### La couche « présentation » ou « vue »

Cette couche est dédiée à l’affichage et aux interactions avec les utilisateurs. Elle sera basée sur le Framework Struts 2 et les JSP (Java Server Page).

* Utilisation de librairies de tags permettant de simplifier le travail du développeur ainsi que la maintenance future : JSTL, DisplayTag…
* Utilisation d’AJAX afin de rendre dynamique les pages web
* Externalisation (et internationalisation possible) des libellés et messages de l’interface en les stockant dans des fichiers textes (1 par langue)
* Gestion des contrôles de surface côté client (rapidité) et serveur (robustesse) via les fichiers XML de configuration de Struts Validator
* Implémentation d’une décoration des JSP (sans développement supplémentaire) à l’aide de Sitemesh (menus, bandeaux…).

### La couche « coordination » ou « contrôleur »

Cette couche coordonne les actions élémentaires pour la réalisation d’un cas d’utilisation. Elle permet de faire le lien entre la vue et le modèle (architecture MVC). Elle sera basée sur le Framework Struts 2.

* Gestion de la navigation
* Gestion efficace de toutes les erreurs fonctionnelles et techniques
* Gestion de la sécurité de façon optimisée
* Purge automatique de la session HTTP avec le principe des conversations

### La couche « service »

Cette couche permet de gérer la façade des objets métiers ainsi que les demandes de services. L’intérêt de cette couche se ressent surtout dans le cas de séparation réelle entre le front et le back-office en termes de déploiement (EJB ou Web Services). Dans l’ENT, nous n’avons pas ce besoin et cette partie est confondue avec la couche domaine.

* Gestion des transactions (et ce quelque soit le type de persistance utilisée)

### La couche « domaine » ou « métier »

Cette couche permet de gérer le modèle métier et les règles de gestion fonctionnelles.

* Remontée des erreurs fonctionnelles (règles de gestion)
* Récupération et sauvegarde des données
* Enchaînements des traitements fonctionnels

### La couche « persistance »

Cette couche gère la persistance des objets. Elle sera basée sur le Framework Ibatis.

* Accés aux données via le modèle de conception DAO. Ainsi la couche persistance pourra être modifiée sans aucun impact sur les autres tiers.

### Architecture en couches et approche Framework

En ayant un rôle bien précis, les composants fournis par chaque couche sont fortement découplés les uns des autres. Ceci présente plusieurs avantages :

* Structuration des composants et amélioration de la qualité
* Développements structurés et totalement modulaires
* Maintenabilité et évolutivité : la modification d’un composant dans une couche n’impacte pas les couches connexes

Afin de permettre aux développeurs de se focaliser entièrement sur le métier, le socle technique fournira les composants adéquats aux différentes problématiques techniques rencontrées sur le projet. Pour cela, divers Frameworks et outils éprouvés dans le domaine des applications J2EE sont utilisés en plus des développements de briques spécifiques.



Les briques du core ENT sont décrites dans la documentation sur les frameworks core (config, cryptage, gestion de fichiers uploadés front+backoffice, échanges entre modules, ftp…). Lorsqu’un besoin similaire apparaît dans le module, ces briques ENT sont à utiliser de préférence à un développement spécifique pour le module ou à d’autres briques externes.

## Architecture applicative globale



En plus des briques techniques tierces vues plus haut, chaque module se base sur un socle commun à tous les modules ENT. Le contenu, l’utilisation, le packaging et le déploiement des différents projets de ce socle sont décrits dans la documentation SPECHTECH-SocleApplicatif.docx.

Nous ne détaillerons pas ici les différentes parties du socle et leurs rôles, mais seulement la façon dont elles s’agencent au sein d’un module.

* Les frameworks core et les apis sont des librairies sous forme de jars embarqués en dépendances dans le war du module.
* Les frameworks sont des utilitaires et des socles techniques utilisées par les différentes couches du module.
* Les apis ne doivent par contre être appelées que depuis la couche métier du module.
* De plus, le module ne connait par défaut que l’api-portail qui sert de façade pour les autres. Il ne doit pas appeler les autres apis directement, mais bien passer par les méthodes des business de l’api-portail qui fait l’intermédiaire selon les règles de gestion adéquates.
* L’api-web-droits est particulière et est utilisée à la fois par le front (vue-contrôleur) et par la couche métier. Les détails de l’intégration de cette api au sein de la webapp sont dans la documentation de l’api-web-droits.
* La partie filtre du client CAS du socle n’est pas embarquée par la webapp. Elle est externalisée et centralisée pour toutes les webapps dans la configuration du serveur Tomcat. Le module n’a aucune dépendance explicite sur CAS, les informations de l’utilisateur lui sont fournies en session (cf documentation sur les frameworks core pour la récupération).

# Création du module

Afin de gagner du temps pour la création du projet, des packages des différentes couches et des configurations par défaut, il faut utiliser le template de projet Maven. L’utilisation est décrite dans la documentation SPECHTECH-SocleApplicatif.docx.

# Règles générales

Les règles générales de développement et le détail des couches backoffice sont dans la documentation SPECHTECH-guide-developpement.docx.

Organisation

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/webapp*: les ressources web (images, js, css, jsp…)
* *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

La gestion des dépendances Maven est décrite dans la documentation sur l’utilisation de Maven dans l’ENT.

Nomenclatures

Les fichiers de configuration de contextes Spring suivent la même nomenclature dans tous les projets ENT. Ils sont placés dans le dossier *spring* de *src/main/resources*. Afin de clarifier la configuration, ils sont découpés en un fichier par couche. On a donc :

* un fichier *actions-applicationContext.xml* pour les beans des actions Struts
* un fichier *business-applicationContext.xml* pour les beans de la couche métier
* un fichier *persistance-applicationContext.xml* pour les beans des DAO et également la configuration des datasources et d’Ibatis
* un fichier *adapter-applicationContext.xml* contenant la configuration de l’adapter Dozer
* un fichier *applicationContext.xml* qui contient les imports de chacun des fichiers précédents, ainsi que la définition des beans transverses, comme le chargement du fichier *config.properties* de l’application. C’est ce fichier seulement qui doit être déclaré dans le *web.xml* de l’application (en plus des fichiers des apis éventuelles).
* Dans le cas d’un module découpé en deux parties (front et back) :
  + Le projet back contiendra les fichiers des couches back (persistance, business, adapter) ainsi qu’un fichier *applicationContext\_back.xml* qui fait les imports de ceux-ci.
  + Le projet front contiendra le fichier *actions-applicationContext.xml*, ainsi que le fichier *applicationContext.xml* qui ne doit alors faire l’import que du fichier actions et du *applicationContext\_back.xml* (en précisant « *classpath:spring/ »* pour le back).

# Développement de la couche contrôleur

## Présentation

La couche contrôleur Struts2 s’appuie sur le modèle MVC2 qui sépare la vue du modèle en les faisant interagir par l’intermédiaire du contrôleur qui gère la navigation.

Struts fournit un contrôleur principal qui redirige vers des sous-contrôleurs. Le contrôleur principal est une servlet unique, qui à l’aide d’un fichier xml (struts.xml) qu’il faut écrire, permet la redirection vers des sous-contrôleurs qu’on appelle Action.

Contrairement à Struts1, avec Struts2 les formulaires qui contiennent les données transmises entre la vue et les sous-contrôleurs, ne sont pas des objets à part mais sont directement les attributs des actions.

Les actions ne doivent pas contenir de traitements métier et de règles de gestion. Elles doivent se contenter de la navigation, de l’enchaînement entre les vues et les méthodes du back-office, le transfert des données avec la session etc…

## Classes de référence

Le package pour la couche contrôleur est le suivant :

*org.lilie.services.<module>.actions*

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

La classe de référence *Action* de chaque module hérite de celle de *fmk-core-web* qui permet de donner accès aux différentes fonctionnalités de Struts en héritant de *ActionSupport*. Elle permet de centraliser les traitements communs éventuels des actions du projet.

Deux attributs y sont pour l’instant centralisés :

* *choixTraitementStr* : il permet à la JSP de préciser une constante correspondant au choix de traitement à effectuer selon l’interaction de l’utilisateur. Cf plus loin le paragraphe sur les actions de fin.
* *provenanceStr* : il permet de renseigner une constante correspondant à la source de l’appel. Cela est utile lorsqu’une page peut être accédée depuis plusieurs autres pages, et que l’on veut pouvoir faire un retour sur la bonne.

## Création des classes actions

### Organisation fonctionnelle

Une action représente en général une fonctionnalité. Plusieurs méthodes dans une même classe action peuvent être utilisées. Cela nous permet donc de regrouper par domaine. En général, on va ainsi diviser par type d’éléments gérés par le module. Par exemple, si un module VidéoClub gère des objets locations et des achats, on crée une classe action pour toutes les méthodes concernant les fonctionnalités à appliquer sur les locations, et une autre pour celles des achats. Bien sûr ceci n’est pas une règle en dur et est adaptable en fonction des besoins du module, de ce qui peut être mutualisé entre les fonctionnalités, des données communes à mettre dans les attributs etc…

### Nommage

Chaque classe Action doit se nommer *<fonctionnalite>Action.java* et être dans le package *org.lilie.services.<module>.actions*, à la racine ou dans un sous-package fonctionnel.

Pour être utilisables en tant qu’action Struts, les méthodes doivent être déclarées *public* et renvoyer un *String* indiquant la suite de la navigation. Les noms de méthode n’ont pas de contrainte mis à part le fait d’être cohérents entre eux, comme précisé pour tout type de méthode dans les règles de nomenclature générale. On commence en général par un verbe correspondant au traitement à effectuer. De préférence, éviter la méthode par défaut « *execute* » et préférer des noms parlants.

### Implémentation

Chaque action doit hériter de la classe *Action* mère du projet.

Les DTOs du back-office peuvent être directement utilisés dans les attributs des actions. Il n’est pas nécessaire de faire un mapping, à part si l’objet utilisé dans la vue a un besoin spécifique différent de celui du métier.

### Spring

Il faut déclarer chaque classe action dans la configuration Spring afin qu’un bean correspondant soit disponible pour l’utilisation par Struts. Dans le fichier *actions-applicationContext.xml*, on ajoute l’élément suivant :

*<bean id="gestionActualiteAction" class="org.lilie.services.actualite.actions.gestion.GestionActualiteAction" scope="prototype"/>*

C’est ensuite le nom du bean, et non le chemin de la classe, qui sera utilisé dans la configuration Struts.

Le scope *prototype* (au lieu de *singleton*) signifie qu’une nouvelle instance sera créée à chaque fois, afin qu’il n’y ait pas de conflits, entre les différents utilisateurs, sur les données échangées dans les attributs du contrôleur.

## Navigation

### Fonctionnement Struts

Chaque action doit être définie dans le fichier *struts.xml* de la manière suivante :

*<action name="adl.message.ajouterMessageAction" class="adlMessageAction" method="ajouterMessage">*

*<result name="input">/jsp/adl/message.jsp</result>*

*<result name="success" type="redirect-action">adl.message.consulterMessagesAction</result>*

*</action>*

Le *name* est le *path* utilisé dans l’url d’appel de l’action.

Dans *class*, on met le nom du bean Spring de l’action, et dans *method* le nom de la méthode pour la fonctionnalité correspondante à cette url.

Pour chaque chaîne de caractères différente renvoyée par l’action, on crée un nouveau *result*, en précisant soit la JSP à afficher ensuite, soit une nouvelle action de redirection.

Pour l’ENT, le nom du path Struts dans l’url doit respecter la nomenclature suivante :

* Mots séparés par des « . »
* On précise dans l’ordre : <domaine>.<sousdomaine>.<fonctionnalite>Action

Avec <domaine>.<sousdomaine> qui va en général correspondre au package et nom de la classe, et la dernière partie qui correspond plus précisément à la méthode.

### Init et Fin

Afin de garder une certaine homogénéité dans l’organisation de la navigation, et ainsi de s’y retrouver plus facilement en passant d’un module à un autre, certaines conventions de navigation ont été définies pour l’ENT.

Pour chaque nouvel écran, on doit avoir une action qui permet d’arriver sur la page (initialisation des données de la page par un appel au back-office par exemple), et une action pour en ressortir. Quelque soit l’interaction de la part de l’utilisateur sur la page, l’action appelée doit toujours être la même action de fin. C’est ensuite cette action de fin qui gère les différentes redirections possibles grâce à l’attribut *choixTraitementStr* de l’action mère, rempli par un champ caché de la JSP.

Concernant le nommage, la méthode et le path de l’action d’arrivée doivent finir par le suffixe « *Init »*, et ceux de l’action de fin par le suffixe « *Fin »*. Les termes d’avant doivent être les mêmes pour l’Init et la Fin.

Cela permet, quelque soit l’écran, de repérer facilement les grands principes de navigation du module simplement par le nommage.

Les actions qui correspondent à des traitements (exporter, enregistrer…) et non à l’affichage d’une page, n’ont pas besoin de suivre ce principe de Init/Fin. Elles sont déjà elles-mêmes appelées par les actions de fin :



Dans l’action de fin, la redirection vers les autres traitements et actions se fait de la manière suivante :

* Si c’est l’appel d’un traitement métier ou d’une méthode action faisant quelques mapping et un appel métier, sans écran particulier correspondant : dans ce cas, on peut appeler directement la méthode.
* Si c’est une redirection vers une autre page: dans ce cas, on repasse par le forward Struts pour appeler l’action d’Init de cette page.

Exemple d’une action de fin :

*public String modifierActualiteFinAction() {*

*try {*

*if (ConstantesActualites.REDIRECTION\_ENREGISTREMENT\_ACTUALITE.equals(getChoixTraitementStr())) {*

*return modifierActualite(); // cas où l’on appelle directement le traitement*

*} else if (ConstantesActualites.REDIRECTION\_UPLOAD\_FICHIER.equals(getChoixTraitementStr())) {*

*return uploadFichier(); // cas où l’on appelle directement le traitement*

*} else if (ConstantesActualites.REDIRECTION\_LECTURE\_ACTUALITE.equals(getChoixTraitementStr())) {*

*return ConstantesActualites.REDIRECTION\_LECTURE\_ACTUALITE; // cas de redirection avec Struts*

*} else if (ConstantesActualites.REDIRECTION\_GESTION\_DROITS.equals(getChoixTraitementStr())) {*

*putConversationDroits(); // cas de redirection avec Struts*

*return ConstantesActualites.REDIRECTION\_GESTION\_DROITS;*

*}*

*return INPUT;*

*} catch (Exception e) {*

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

*return ERROR;*

*}*

*}*

## Gestion des exceptions

La gestion des exceptions à mettre en place est décrite dans son ensemble dans la documentation des frameworks core.

Voici un exemple cependant d’un catch assez complet dans une action :

*} catch (ServiceFonctionnelleException sfe) {*

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

*if (ConstantesActualites.CODE\_ERREUR\_ACTU\_ABSENTE.equals(sfe.getErrorCode())) {*

*addActionError(getText(ConstantesI18n.ACTUALITE\_NON\_DISPONIBLE\_MESSAGE));*

*return ConstantesActualites.REDIRECTION\_ACTUALITE\_ABSENTE;*

*} else {*

*addActionError(sfe.getMessage());*

*return INPUT;*

*}*

*} catch (ServiceTechniqueException ste) {*

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

*return ERROR;*

*} catch (Exception e) {*

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

*return ERROR;*

*}*

On voit bien que les exceptions fonctionnelles sont rattrapées de manière spécifique, avec un test explicite sur le code erreur pour afficher le bon message à l’utilisateur (+ un traitement d’affichage par défaut au cas où). Les erreurs techniques sont quant à elles bien renvoyées vers la page d’erreur.

Ou plus exactement, elles sont renvoyées vers le forward Struts *ERROR*, et celui-ci est mappé dans le *struts.xml* pour renvoyer vers la page d’erreur :

*<global-exception-mappings>*

*<exception-mapping result="nosuchmethod" exception="java.lang.NoSuchMethodException" />*

*<exception-mapping result="actionexception" exception="org.lilie.socle.core.business.exception.ActionTechniqueException" />*

*<exception-mapping result="unhandledexception" exception="java.lang.Exception" />*

*<exception-mapping result="sizeexception" exception="org.apache.commons.fileupload.FileUploadBase$SizeLimitExceededException" />*

*<exception-mapping result="accesnonautoriseexception" exception="org.lilie.socle.core.utils.exception.AccesNonAutoriseException" />*

*</global-exception-mappings>*

*<global-results>*

*<result name="nosuchmethod">/jsp/error/error.jsp</result>*

*<result name="actionexception">/jsp/error/error.jsp</result>*

*<result name="error">/jsp/error/error.jsp</result>*

*<result name="unhandledexception">/jsp/error/error.jsp</result>*

*<result name="accesnonautoriseexception">/jsp/error/accesNonAutorise.jsp</result>*

*</global-results>*

## Validation des données

### Principe

Les règles de gestion et contrôles métier sont à effectuer de préférence dans la couche métier avec le renvoi d’exceptions fonctionnelles comme vu plus haut.

Cependant, afin de rendre l’interface plus réactive et d’éviter les allers-retours serveur inutiles, certaines validations peuvent être faites côté client en javascript. Ceci concerne seulement les contrôles de saisie simples : champ obligatoire, type de saisie (chaîne, entier, date…). Par contre cela n’empêche pas de conserver également un contrôle côté serveur. Le back-office doit toujours maîtriser ce qui lui arrive en entrée et donc vérifier que les données sont dans le bon format.

Pour ces validations, nous utilisons le mécanisme proposé par Struts2, qui permet d’activer les deux contrôles à la fois.

### Validator Struts

Le Validator Struts permet d’appliquer des contrôles sur les champs d’un formulaire :

* Dans *src/main/resources*, dans la même arborescence que le package de l’action concernée, créer un fichier *<NomClassAction>-<NomPathStruts>-validation.xml* et un fichier *<NomClasseAction>.properties* qui contiendra les libellés des messages d’erreur. Les libellés peuvent également être centralisés dans le fichier messages.properties du module.
* Dans le fichier xml de validation, on précise alors pour chaque champ le contrôle à effectuer :

*<validators>*

*< !-- le nom du champ qui correspond à un attribut de la class action -->*

*<field name="adlMessageDto.valeurStr">*

*< !-- le contrôle à effectuer -->*

*<field-validator type="requiredstring">*

*< !-- la clé dans le fichier properties associé -->*

*<message key="erreur.obligatoire.message"/>*

*</field-validator>*

*</field>*

*</validators>*

* Il faut ensuite ajouter l’activation de la validation dans le formulaire de la JSP :

*<s:form method="post" action="adl.message.ajouterMessageAction.action"* ***validate="true" theme="css\_xhtml"****>*

Afin que cela fonctionne mieux, il est conseillé de modifier le thème de génération des balises HTML par Struts en *css\_xhtml*.

* La méthode *validate()* de chaque classe action, qui est appelée avant le traitement de l’action, peut éventuellement être utilisée pour des contrôles ne pouvant être faits dans le xml. Mais dans ce cas, il n’y aura pas de validation côté client. Dans tous les cas, préférer le xml et les exceptions fonctionnelles de la couche métier.

## Conversations

La gestion des données en session doit s’appuyer sur le socle proposé par le projet fmk-core-web. La description de son utilisation et de son fonctionnement interne sont dans la documentation sur les frameworks core.

Afin que le filtre de gestion des conversations soit activé, il faut le déclarer dans le fichier *struts.xml* de la webapp :

*<interceptors>*

*<interceptor name="conversation" class="org.lilie.socle.core.web.actions.conversation.ConversationInterceptor" />*

*<interceptor-stack name="org.lilie.defaultstack">*

*<interceptor-ref name="conversation" />*

*<interceptor-ref name="defaultStack" />*

*</interceptor-stack>*

*</interceptors>*

*<default-interceptor-ref name="org.lilie.defaultstack" />*

## Sécurité

### Objectif

Les droits d’accès aux différentes fonctionnalités sont vérifiés dans l’interface afin de n’afficher que ce à quoi l’utilisateur est habilité. Cependant, bien que cela réponde au besoin fonctionnellement parlant, ce n’est pas suffisant en termes de sécurité. En effet, bien que l’interface ne le propose pas, un utilisateur malveillant pourrait changer les urls et les paramètres envoyés pour essayer de passer outre ses droits. C’est pour cela qu’il est nécessaire de faire systématiquement une vérification côté serveur.

Les habilitations ne sont pas seulement par fonctionnalité et par type de profil. Elles sont plus précises que cela, par utilisateur et par donnée. C’est pourquoi l’utilisation d’un framework standard générique comme Spring Security n’aurait pas suffit pour répondre au besoin. Nous avons donc choisi de développer une solution spécifique centralisée aux projets ENT.

### Solution choisie

La gestion de la sécurité se base sur la programmation par aspects de Spring (Spring AOP). Cela permet d’intercepter l’appel de certaines méthodes et de définir des traitements qui seront lancés systématiquement avant.

Afin de préciser quelle méthode doit être interceptée, nous utilisons des annotations.

Des aspects plus ou moins génériques propres aux différents cas d’utilisation fonctionnels des ENT ont été développés dans le socle. Ceux concernant les droits sur les données de l’annuaire sont dans l’api-portail/api-annuaire. Ceux concernant les droits sur les entités des modules sont définis dans l’api-web-droits.

L’utilisation de ces aspects est à faire dans la couche contrôleur qui est la première accédée par la vue côté client. C’est donc au niveau des classes actions que l’on doit préciser quels contrôles effectuer.

### Utilisation par le module

Annotations

A chaque aspect du socle correspond une annotation pour l’utiliser. Il suffit de la mettre sur la méthode action voulue. Plusieurs paramètres peuvent être passés à l’annotation. Ceux-ci sont visibles dans la javadoc de l’annotation.

*@AutoriseEntiteLecture(methodeEntiteId="getEntiteIdLecture", methodePagePubliqueBln="isPubliqueBln")*

*public String lectureActualiteInitAction() {*

*…*

*}*

Il est possible d’avoir plusieurs annotations de sécurité par méthode :

*@AutoriseEntiteLecture(methodeEntiteId="getEntiteIdLecture",*

*methodePagePubliqueBln="isPubliqueBln", listeTraitementStr={ConstantesActualites.REDIRECTION\_AJOUT\_COMMENTAIRE,ConstantesActualites.REDIRECTION\_NOTIFICATION\_ACTUALITE, ConstantesActualites.REDIRECTION\_SUPPRESSION\_NOTIFICATION\_ACTUALITE})*

*@AutoriseEntiteGestion(methodeEntiteId="getEntiteIdLecture",*

*listeTraitementStr={ConstantesActualites.REDIRECTION\_CACHE\_COMMENTAIRE, ConstantesActualites.REDIRECTION\_MONTRE\_COMMENTAIRE})*

*@Autorise(methodeControle="isAutoriseSuppressionCommentaire",*

*listeTraitementStr={ConstantesActualites.REDIRECTION\_SUPPRESSION\_COMMENTAIRE})*

*public String lectureActualiteFinAction() throws AccesNonAutoriseException {*

*….*

*}*

Si l’on ne veut pas faire tous les contrôles systématiquement, par exemple pour une méthode action de Fin, où le type de contrôle à faire dépend du traitement vers lequel on va être redirigé :

Utiliser le paramètre *listeTraitementStr* qui permet de lister les *choixTraitementStr* pour lesquels on applique le contrôle.

Exceptions

Les aspects renvoient des exceptions de sécurité quand le contrôle n’est pas valide.

Les méthodes ayant des annotations doivent donc ajouter à leur signature le *throws* adéquat :

*public String lectureActualiteInitAction() throws AccesNonAutoriseException {*

*…*

*}*

Dans le *struts.xml*, il faut également avoir la définition du *catch* pour rediriger sur la page d’erreur spécifique aux droits :

<global-results>

<result name="accesnonautoriseexception">/jsp/error/accesNonAutorise.jsp</result>

</global-results>

<global-exception-mappings>

<exception-mapping result="accesnonautoriseexception" exception="org.lilie.socle.core.utils.exception.AccesNonAutoriseException" />

</global-exception-mappings>

Il faut ensuite ajouter le libellé du message d’erreur dans le fichier *messages.properties* du module:

*error.accesnonautorise=Vous n''avez pas les droits nécessaires pour accéder à cette page.*

Sur quelles méthodes ?

Pour chaque nouvelle action, il faut penser à ajouter les annotations de sécurité nécessaires :

* Sur les méthodes Init
* Sur les méthodes Fin :

Test seulement pour les appels qui vont vers des traitements directs, les appels qui enchaînent sur un autre Init n'ont pas besoin d'être vérifiés car ils le seront par l'Init elle-même.

* Sur les autres méthodes: si pas dans le *struts.xml*, rien à faire.
* Les onglets de droits et permissions centralisés dans l’api-web-droits sont déjà sécurisés.
* Pas sur les « sous-méthodes » : On peut mettre les aspects seulement sur des méthodes appelées directement depuis le bean action Spring, c'est-à-dire les méthodes actions appelées directement par Struts. On ne peut pas mettre d’aspect sur une méthode d’une classe qui serait appelée elle-même par une autre méthode de la même classe. C’est pourquoi on est obligé de mettre les annotations sur les méthodes *Fin*, avec la liste de *choixTraitementStr*.

Concernant les paramètres, « Utilisateur » et « methodeChoixTraitement » ne sont pas utiles pour les webapps classiques Lilie car ils sont récupérés de manière générique par défaut.  
Par contre pour les portlets ou la console d’admin, la récupération de l'utilisateur n'étant pas pareille, il faut forcément utiliser le paramètre proposé.

Besoin spécifique

Si un besoin de contrôle particulier n’est pas présent dans le socle :

* Se poser la question de l’utilité ou non d’en faire un nouvel aspect/annotation centralisé dans le socle, ou bien de faire évoluer un existant avec un paramètre ou autre.
* En dernier recours, il existe dans le socle un aspect/annotation générique qui permet de fournir en paramètre le nom d’une méthode de l’action qui fera le contrôle elle-même. Le traitement du contrôle est donc dans ce cas à faire spécifiquement pour cette action.

### Fonctionnement interne

Pour définir un aspect, il faut ajouter la configuration Spring suivante :

* Ajouter ceci pour que le proxy se fasse automatiquement sur les méthodes voulues :

*<aop:aspectj-autoproxy proxy-target-class="true"/>*

* Dans la balise *<aop:config>,* préciser quelle méthode de quelle classe fait le traitement de contrôle de l’aspect :

*<aop:aspect id="isAutoriseCreation" ref="securiteEntite">*

*<aop:around method="isAutoriseCreation" pointcut-ref="autoriseEntiteCreationPointCut"/> </aop:aspect>*

*securiteEntite* étant le bean Spring représentant la classe où il y a le traitement

* Dans la balise *<aop:config>,* préciser quelles méthodes doivent être interceptées pour ce contrôle (ici par annotation) :

*<aop:pointcut id="autoriseEntiteCreationPointCut" expression="@annotation(org.lilie.socle.api.webdroits.securite.annotations.AutoriseEntiteCreation)"/>*

L’annotation Java doit être définie avec les annotations suivantes:

*@Inherited*

*@Retention(RetentionPolicy.RUNTIME)*

*@Target({ElementType.TYPE, ElementType.METHOD })*

La méthode de l’aspect doit prendre en paramètre: *ProceedingJoinPoint joinPoint.*

A la fin du traitement de contrôle:

* Si ok, lancer l’action : *return (String) joinPoint.proceed();*
* Sinon, lancer une exception de type *AccesNonAutoriseException.*

# Développement de la couche vue

## Découpage

La vue est représentée principalement par des JSP et les ressources statiques associées. Toutes ces ressources sont dans le dossier *src/main/webapp* du module.

Les JSP sont dans le dossier « jsp », les ressources Javascript dans le dossier « js », les images dans le dossier « img », les feuilles de style dans le dossier « css ». Les fichiers de configuration de la vue et de la webapp sont dans le dossier « WEB-INF ».

Le découpage dans chaque dossier est ensuite un découpage fonctionnel.

## Javascript

Aucun code Javascript ne doit être dans la JSP. Il doit être au maximum externalisé dans des fichiers à part. Si ce sont des fonctions spécifiques pour une page, le fichier JS doit avoir le même nom que le fichier JSP, et être dans la même arborescence fonctionnelle dans le dossier « js ». Si ce sont des fonctions communes à plusieurs pages, elles doivent être dans un fichier dans un dossier « commun » aux plusieurs pages, voire même dans le fmk-core-web si c’est partageable avec d’autres modules.

Les inclusions de fichiers Javascript dans les JSP doivent être en fin de page pour ne pas retarder l’affichage de la page (à moins que l’une des fonctions soit nécessaire au chargement).

Comme pour le code Java, il faut bien commenter le code Javascript : rôle des méthodes, paramètres, et commentaires dans les traitements car le Javascript n’est pas un langage facile à lire et à maintenir.

Afin de rendre les pages plus dynamiques, il est possible d’utiliser des appels Ajax, par l’intermédiaire des méthodes de la librairie JQuery (pour l’utilisation au sein de l’ENT, voir dans la documentation fmk-core-web).

## JSP

* De manière générale, un écran correspond à une JSP. Il est possible d’avoir des inclusions de JSPF afin de rendre les écrans plus modulaires et de factoriser des éléments.
* Le code doit être indenté et tous les libellés doivent être externalisés dans le fichier chargé par Struts : *messages.properties*.
* Les balises à utiliser sont les balises fournies par Struts en priorité, puis la libraire JSTL si besoin. Il faut éviter le code Java directement dans la vue.
* Les formulaires doivent être envoyés en POST autant que possible (sauf contraintes dues au portail Liferay).
* Pour l’affichage des listes, utiliser au maximum la libraire Displaytag qui permet d’encapsuler beaucoup de choses (style, tris, pagination, formattage…).

Le look du displaytag est défini dans un fichier de configuration : *displaytag.properties*.

Voici la configuration par défaut avec les styles de l’ENT :

locale.provider=org.lilie.services.blog.utils.I18nEntAdapter

locale.resolver=org.lilie.services.blog.utils.I18nEntAdapter

#locale.provider=org.displaytag.localization.I18nStrutsAdapter

#=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

# Messages par défaut

#=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

basic.msg.empty\_list=Aucun résultat ne correspond à vos critères de recherche.

error.msg.invalid\_page=Message invalide, veuillez contactez l'administrateur.

#=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

# Pagination et nombre de résultats

#=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

paging.banner.item\_name=blog&nbsp;

paging.banner.items\_name=blogs&nbsp;

paging.banner.no\_items\_found=<span>Aucune {0} trouvée</span>

paging.banner.one\_item\_found=

paging.banner.all\_items\_found=

paging.banner.some\_items\_found=

paging.banner.full = <div id="appli\_pagination-header"><div class="sortpage" id="zoneNombrePage"><span>Afficher </span><a title="5" href="#" onclick="changerNombreBlogParPage(''blogForm'',5);" name="nbItem\_5"><b>5</b></a><a href="#" onclick="changerNombreBlogParPage(''blogForm'',10);" title="10" name="nbItem\_10"><b>10</b></a><a title="25" href="#" onclick="changerNombreBlogParPage(''blogForm'',25);" name="nbItem\_25"><b>25</b></a><a title="50" href="#" onclick="changerNombreBlogParPage(''blogForm'',50);" name="nbItem\_50"><b>50</b></a><span title="résultats par page"> résultats par page</span></div><a href="#" onClick="allerVersPage(''{2}''); return false;" title="Aller à la page précédente">Page précédente</a> {0} <a href="#" onClick="allerVersPage(''{3}''); return false;" title="Aller à la page suivante"> Page suivante</a><div class="clearboth"> </div></div>

paging.banner.first = <div id="appli\_pagination-header"><div class="sortpage" id="zoneNombrePage"><span>Afficher </span><a title="5" href="#" onclick="changerNombreBlogParPage(''blogForm'',5);" name="nbItem\_5"><b>5</b></a><a href="#" onclick="changerNombreBlogParPage(''blogForm'',10);" title="10" name="nbItem\_10"><b>10</b></a><a title="25" href="#" onclick="changerNombreBlogParPage(''blogForm'',25);" name="nbItem\_25"><b>25</b></a><a title="50" href="#" onclick="changerNombreBlogParPage(''blogForm'',50);" name="nbItem\_50"><b>50</b></a><span title="résultats par page"> résultats par page</span></div> {0} <a href="#" onClick="allerVersPage(''{3}''); return false;" title="Aller à la page suivante">Page suivante</a><div class="clearboth"> </div></div>

paging.banner.last = <div id="appli\_pagination-header"><div class="sortpage" id="zoneNombrePage"><span>Afficher </span><a title="5" href="#" onclick="changerNombreBlogParPage(''blogForm'',5);" name="nbItem\_5"><b>5</b></a><a href="#" onclick="changerNombreBlogParPage(''blogForm'',10);" title="10" name="nbItem\_10"><b>10</b></a><a title="25" href="#" onclick="changerNombreBlogParPage(''blogForm'',25);" name="nbItem\_25"><b>25</b></a><a title="50" href="#" onclick="changerNombreBlogParPage(''blogForm'',50);" name="nbItem\_50"><b>50</b></a><span title="résultats par page"> résultats par page</span></div><a href="#" onClick="allerVersPage(''{2}''); return false;" title="Aller à la page précédente">Page précédente</a> {0} <div class="clearboth"> </div></div>

paging.banner.page.link=<a href="#" onClick="allerVersPage(''{1}''); return false;" title="Aller à la page {0}">{0}</a>

paging.banner.page.selected=<span class="bold">{0}</span>

paging.banner.onepage= <div id="appli\_pagination-header"><div class="sortpage" id="zoneNombrePage"><span>Afficher </span><a title="5" href="#" onclick="changerNombreBlogParPage(''blogForm'',5);" name="nbItem\_5"><b>5</b></a><a href="#" onclick="changerNombreBlogParPage(''blogForm'',10);" title="10" name="nbItem\_10"><b>10</b></a><a title="25" href="#" onclick="changerNombreBlogParPage(''blogForm'',25);" name="nbItem\_25"><b>25</b></a><a title="50" href="#" onclick="changerNombreBlogParPage(''blogForm'',50);" name="nbItem\_50"><b>50</b></a><span title="résultats par page"> résultats par page</span></div> {0} page<div class="clearboth"> </div></div>

paging.banner.page.separator=&nbsp;

paging.banner.placement=top

sort.amount=list

sorting.page.link=<a href="javascript:allerVersPage(''{1}''); return false;">{0}</a>

css.tr.odd=

css.tr.even=

## Images et CSS

Les images et CSS spécifiques au module peuvent être stockées dans le module. Aucun style ne doit apparaître en dur directement dans les balises JSP. Les styles doivent être externalisés en feuilles CSS dans des fichiers à part.

Les inclusions des feuilles de style doivent se faire en début de JSP afin de permettre à la page de s’afficher plus rapidement.

Dans tous les cas, il faut limiter la taille des images pour éviter des pages trop lourdes à charger.

## Ressources du fmk-core-web

Les ressources (images, css, js, tld…) des modules sont pour la plupart communes. Elles doivent respecter la charte ENT. Afin de centraliser les besoins fonctionnels et techniques, ces ressources sont mutualisées. Au lieu d’être présentes dans chaque projet de module, elles sont stockées dans le projet fmk-core-web. La dépendance du module sur le projet fmk-core-web, permet à la webapp, au moment du packaging, de récupérer les ressources pour les ajouter au war final.

La description de ces ressources communes et leur utilisation en termes de packaging sont précisées dans la documentation sur les frameworks core.

Concernant l’appel d’une ressource dans une JSP, il ne faut pas oublier de préciser le chemin adéquat : lors du packaging, les ressources du core suivent le découpage précisé plus haut et sont placées dans un sous-dossier « core ».

## Sitemesh

### Objectif

Afin d’homogénéiser les pages et de centraliser l’affichage des éléments communs, on utilise le principe de modèles de page. On définit un template que l’on peut appliquer aux contenus des pages souhaitées. Cela permet d’ajouter la décoration des pages (header, footer, bandeaux….) en ne les définissant qu’une seule fois.

Pour cela, nous utilisons l’outil Sitemesh.



### Définition du template

Le modèle de page et les éléments de décoration à y mettre sont à placer dans le dossier :

*src/main/webapp/jsp/decoration.*

Les modèles de pages sont des JSP et les éléments qu’il contient sont à mettre dans des JSPF.

On peut avoir plusieurs modèles selon les différents domaines fonctionnels de l’application (main.jsp, mainWysiwyg.jsp, modeleAccueil.jsp…).

Contenu du modèle :

* Ajouter les taglibs Sitemesh :

*<%@taglib prefix="decorator" uri="http://www.opensymphony.com/sitemesh/decorator" %>*

*<%@taglib prefix="page" uri="http://www.opensymphony.com/sitemesh/page" %>*

* Dans le *<head>*, ajouter les inclusions de feuilles de styles communes ainsi que les fichiers Javascript utilisés par toutes les pages utilisant ce modèle. Cela permet de ne pas avoir à les redéfinir dans la JSP de chaque écran.
* A la fin du *<head>* du modèle, ajouter ensuite la balise <*decorator:head>.* Cela précise à Sitemesh de récupérer également les éléments spécifiques définis dans le *<head>* de la page elle-même.
* La balise *<decorator:title>* permet de définir le titre de la page dans le navigateur en utilisant celui précisé dans chaque page (balise *title*), ou bien un titre par défaut si non renseigné dans la page.
* Dans le *<body>*, on peut ensuite mettre ce que l’on veut. C’est là que l’on définit la mise en page et que l’on place chaque élément (bandeau, contenu etc…) à l’aide des balises HTML classiques (et des balises Struts éventuellement).
* Pour inclure un élément JSPF dans le <body>, il suffit d’ajouter la ligne suivante :

*<jsp:directive.include file="jspf/bandeau.jspf"/>*

* Pour placer le corps de la page, c'est-à-dire la JSP qui utilise ce modèle, il faut utiliser la balise *<decorator:body />.*

Toute la partie *<body>* de la JSP sera incluse en remplacement de cette balise dans le modèle.

Voici un exemple de modèle assez générique à utiliser pour l’ENT :

*<!DOCTYPE html PUBLIC*

*"-//W3C//DTD XHTML 1.1 Transitional//EN"*

*"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">*

*<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>*

*<%@taglib prefix="decorator" uri="http://www.opensymphony.com/sitemesh/decorator" %>*

*<%@taglib prefix="page" uri="http://www.opensymphony.com/sitemesh/page" %>*

*<%@taglib prefix="s" uri="/struts-tags" %>*

*<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" lang="fr">*

*<head>*

*<link rel="stylesheet" href="<s:url value="css/ENTIDF\_webapp.css" includeParams="false"/>" />*

*<link rel="stylesheet" href="<s:url value="css/ENTIDF\_webapp\_structure\_sans\_menu.css" includeParams="false"/>" />*

*<script type="text/javascript" src="<s:url value="js/core/ajax.js" includeParams="false"/>"></script>*

*<script type="text/javascript" src="<s:url value="js/core/slider.js" includeParams="false"/>"></script>*

*<script type="text/javascript" src="<s:url value="js/actualite.js" includeParams="false"/>"></script>*

*<script type="text/javascript" src="<s:url value="js/core/liste.js" includeParams="false"/>"></script>*

*<title><decorator:title default="WebappENTIDF"/></title>*

*<decorator:head/>*

*</head>*

*<body>*

*<div id="global">*

*<!-- haut de page -->*

*<div class="page">*

*<div class="viewer">*

*<div id="main">*

*<decorator:body />*

*</div>*

*</div>*

*</div>*

*</div>*

*</body>*

*</html>*

### Utilisation pour les pages

Le fichier *decorators.xml*, placé dans *WEB-INF*, permet de préciser quelles pages utilisent quel modèle. Ce sont les URLs avec les path Struts que l’on doit préciser et non directement les JSP. On peut utiliser des pattern pour les urls. On peut également utiliser des exclusions. Il faut notamment bien penser à exclure les actions de type Ajax et les urls des ressources statiques.

*<?xml version="1.0" encoding="ISO-8859-1"?>*

*<decorators defaultdir="/jsp/decoration">*

*<excludes>*

*<pattern>/nodecorate/\*</pattern>*

*<pattern>/css/\*</pattern>*

*<pattern>/js/\*</pattern>*

*<pattern>/img/\*</pattern>*

*<pattern>/dojo/\*</pattern>*

*<pattern>/struts/\*</pattern>*

*<pattern>/affichagePageErreur\*</pattern>*

*<pattern>/\*error\*</pattern>*

*<pattern>/\*Ajax\*</pattern>*

*<pattern>/Supervision\*</pattern>*

*</excludes>*

*<decorator name="mainDroitsActualites" page="mainDroitsActualites.jsp">*

*<pattern>/modifierDroitsEtPermissionsInit.action</pattern>*

*</decorator>*

*<decorator name="mainWysiwyg" page="mainWysiwyg.jsp">*

*<pattern>/modificationActualite\*</pattern>*

*<pattern>/creationActualite\*</pattern>*

*<exclude>/jsp/error/error.jsp</exclude>*

*</decorator>*

*<decorator name="main" page="main.jsp">*

*<pattern>/\*</pattern>*

*<exclude>/jsp/error/error.jsp</exclude>*

*</decorator>*

*</decorators>*

### Configuration de la webapp

Afin d’activer Sitemesh dans le webapp, il faut ajouter le filtre suivant dans le *web.xml* :

*<filter>*

*<filter-name>sitemesh</filter-name>*

*<filter-class>com.opensymphony.module.sitemesh.filter.PageFilter</filter-class>*

*</filter>*

*<filter-mapping>*

*<filter-name>sitemesh</filter-name>*

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

*</filter-mapping>*

Le fichier *sitemesh.xml*, placé dans *WEB-INF*, permet de configurer Sitemesh. Pour l’ENT, nous avons gardé les paramétrages par défaut.

## Web.xml

Le fichier *web.xml* de la webapp doit contenir :

* les différents filtres des frameworks utilisés (Spring, Struts2, Sitemesh)

*<listener>*

*<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>*

*</listener>*

*<filter>*

*<filter-name>action2-cleanup</filter-name>*

*<filter-class>org.apache.struts2.dispatcher.ActionContextCleanUp</filter-class>*

*</filter>*

*<filter>*

*<filter-name>sitemesh</filter-name>*

*<filter-class>com.opensymphony.module.sitemesh.filter.PageFilter</filter-class>*

*</filter>*

*<filter>*

*<filter-name>action2</filter-name>*

*<filter-class>org.lilie.services.actualite.actions.reference.EntFilterDispatcher</filter-class>*

*</filter>*

*<filter-mapping>*

*<filter-name>action2-cleanup</filter-name>*

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

*</filter-mapping>*

*<filter-mapping>*

*<filter-name>sitemesh</filter-name>*

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

*</filter-mapping>*

*<filter-mapping>*

*<filter-name>action2</filter-name>*

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

*</filter-mapping>*

* les imports de contextes Spring

*<context-param>*

*<param-name>contextConfigLocation</param-name>*

*<param-value>*

*classpath:spring/applicationContext.xml*

*classpath:spring/applicationContext\_api-portail.xml*

*classpath:spring/applicationContext\_fmk-core-web.xml*

*classpath:spring/applicationContext\_api-web-droits.xml*

*</param-value>*

*</context-param>*

* éventuellement la déclaration des services REST si la webapp en a (cf documentation des frameworks core pour utilisation)
* la servlet de supervision

*<servlet>*

*<display-name>MaintenanceServlet</display-name>*

*<servlet-name>MaintenanceServlet</servlet-name>*

*<servlet-class>org.lilie.services.actualite.utils.MaintenanceServlet</servlet-class>*

*</servlet>*

*<servlet-mapping>*

*<servlet-name>dwr</servlet-name>*

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

*</servlet-mapping>*

*<servlet-mapping>*

*<servlet-name>MaintenanceServlet</servlet-name>*

*<url-pattern>/Supervision</url-pattern>*

*</servlet-mapping>*

# Constantes et utilitaires

En plus des packages des différentes couches, 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*.

# Packaging

Les modules sont des webapps et sont donc packagés dans des archives WAR. Le WAR contient toutes les dépendances (externes et socle ENT) du module.

Les fichiers de configuration doivent être externalisés (cf documentation sur les frameworks core pour la mise en place de ce fonctionnement).

A chaque build, le module doit récupèrer les ressources du fmk-core-web et de l’ap-web-droits dans la bonne version pour les ajouter au WAR (configuration et utilisation précisées dans le détail de la couche vue de cette documentation et dans la documentation sur les frameworks core).

Si le module est en deux parties (front et back), dans ce cas, le projet back est packagé dans un JAR, et le front a une dépendance sur ce JAR qui est alors embarqué dans le WAR comme les autres librairies.

# 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 |