LUTECE : GUIDE DÉVELOPPEUR

Published : 2015-09-15
License : GPL

INTRODUCTION

QU'EST CE QUE LUTECE ?

Lutece est un **logiciel libre** diffusé sous **licence BSD** par la Mairie de Paris depuis septembre 2002.

Lutece est écrit en **Java** et repose sur des briques logicielles Open Source faisant partie des standards (SpringFramework, Lucene, Freemarker, Ehcache, Apache-Commons, ...).
Il a été accepté en 2014 au sein du consortium Open Source international OW2.

Lutece était à l'origine un CMS / Portail, mais a évolué au fil des années vers un Framework de développement d'applications Web. Cet outil est très largement utilisé par la Mairie de Paris pour la réalisation de ses sites, ses services numériques et ses applications métiers.

Lutece est ou a été utilisé par de prestigieuses références telles que MétéoFrance.fr, Notaires.fr, AtosWorldline.com, GeoPortail.gouv.fr, Paris.fr ou Marseille.fr.

L'architecture très modulaire de **Lutece** permet une très grande souplesse dans l'intégration de fonctionnalités spécifiques à un portail. Les plugins, les modules, l'injection de dépendances sont autant de solutions pour réaliser des interfaces sur mesure tant sur le *Front Office* que sur le *Back Office*.

Pour compléter votre information au-delà de ce livre et rester en contact, voici une liste de liens à mettre dans vos favoris :

- Le site officiel : http://fr.lutece.paris.fr
- La démo en ligne (données réinitialisées toutes les 3 heures) : http://dev.lutece.paris.fr/site-demo/
- Le code source : https://github.com/lutece-platform
- Les versions eBook de ce présent livre et des manuels utilisateur et intégrateur : http://www.flossmanualsfr.net
- Les métriques Open Source du projet : https://www.openhub.net/p/lutece
- La documentation Maven du projet : http://dev.lutece.paris.fr
- Le Wiki contenant les Howto : http://wiki.lutece.paris.fr
- Le compte twitter pour les news générales : https://twitter.com/LuteceNews
- Le compte twitter pour les news de la fonderie : https://twitter.com/LuteceFoundry
- Le site Meetup : http://www.meetup.com/Lutece/
- L'image Docker du site de démo : https://registry.hub.docker.com/u/lutece/lutece-demo/

À PROPOS DE CE LIVRE

Ce livre se destine à des développeurs souhaitant créer des nouvelles fonctionalités basées sur une plate-forme **Lutece**. Il peut s'agir d'applications Front Office telles que des téléservices, d'applications métiers utilisant le Back Office de Lutece ou encore d'étendre les fonctions existantes du portail.

La première partie du livre traite des aspects théoriques : l'architecture logicielle, les normes de developpement et les principaux frameworks utilisés.
La deuxième partie, quant à elle, est consacrée à des guides pratiques de développement.

Nous parlerons ici de la version 5.1 de Lutece.

LES AUTEURS

Ce livre a été rédigé durant un Booksprint de 5 jours en août 2015, organisé par Floss Manuals Francophone et l'équipe de **Lutece**. Les coauteurs de cet ouvrage sont : Magali Bouvat, Baptiste Gabourin, Cédric Gémy, Jon Harper, Laurent Hohl, Elysa Jouve, Magali Lemaire, Isabelle Lenain, Pierre Levy, Alexandre Mangot, Loïc Martinez, François-Eric Merlin, Yvan Michel, Jérome Poirier, Marie-Pierre Roux, Élisabeth Saumard, Rémi Zara et Elisa de Castro Guerra.

Certains contenus proviennent de contributions plus anciennes réalisées par : Valérian Broussard, Géraud Dugé de Bernonville, Thibault Lassalle, Louis Lin, Vashista Nobaub, Vincent Vossier, Johann Goulay.

LICENCE

Cet ouvrage est sous licence **GPL** et licence **Creative Commons By SA**. Vous êtes libre de le lire, le partager, le modifier.

PRÉSENTATION GÉNÉRALE DE LUTECE

1. DESCRIPTION D'UNE WEBAPP LUTECE
2. DESCRIPTION FONCTIONNELLE DES COMPOSANTS
3. DESCRIPTION DE L'ARCHITECTURE LOGICIELLE
4. DESCRIPTION DE LA COMPOSITION DES PLUGINS

1. DESCRIPTION D'UNE WEBAPP LUTECE

Dans ce chapitre, nous allons décrire la philosophie d'organisation d'une webapp lutece de manière très simplifiée. Les notions seront ensuite approfondies dans la suite du livre. Il s'agit ici de donner des repères au lecteur qui aideront la compréhension des chapitres suivants.

FRONT OFFICE / BACK OFFICE

Lorsqu'une webapp est déployée dans un conteneur de servlet, l'intégralité des fonctions du site sont accessibles depuis les URLs de cette webapp, par exemple *http://www.example.com/webapp-example/* que l'on abregera *<webapp>* dans la suite de ce chapitre.

On distingue d'emblée dans Lutece deux parties qui sont totalement séparées:

 * Le front office, destiné au public : URLs sous *<webapp>/jsp/site/*

 * Le back office, destiné aux administrateurs : URLs sous *<webapp>/jsp/admin/*

LE FRONT OFFICE

Le point d'entrée principal: Portal.jsp

La quasi totalité des contenus html du Front Office sont accessibles à l'URL *<webapp>/jsp/site/Portal.jsp*. Le contenu de la page est determiné par les paramètres de la requete HTTP. Par exemple:

- *<webapp>/jsp/site/Portal.jsp?page_id=13*
- *<webapp>/jsp/site/Portal.jsp?page=map*
- *<webapp>/jsp/site/Portal.jsp?document_id=21&portlet_id=1*

Les différents parametres HTTP servent à utiliser différents **Content Services** que nous découvriront plus tard dans ce livre. Lorsqu'il n'y a pas de paramètres HTTP, c'est le PageContentService ("page_id=XX") qui est utilisé et qui renvoie le contenu de sa page racine (par défaut page_id=1, mais configurable).

Note: ces URLs techniques peuvent être remplacées par des URLs explicites en utilisant le plugin-seo (exemple: *<webapp>/home.html* pour *<webapp>/jsp/site/Portal.jsp?page_id=1*, ou *<webapp>/sitemap.html* pour *<webapp>/jsp/site/Portal.jsp?page=map*)

Autres URLs classiques

- Lors de l'acces à l'URL *<webapp>/*, le contenu renvoyé est celui d'une URL de la webapp, configurable en Back Office. Par défaut, cette URL est <webapp>/jsp/site/Portal.jsp.
- Les ressources statiques sont accessibles directement sous *<webapp>/images/**, *<webapp>/css/** ou *<webapp>/js/**.
- Les web services REST sont accessibles sous l'URL <webapp>/rest/* (nécéssite a minima le **plugin-rest** et un plugin publiant des web services, comme **module-document-rest**).

LE BACK OFFICE

L'accueil du back office est accessible à l'URL *<webapp>/jsp/admin/AdminMenu.jsp*. Contrairement au contenus du Front Office accessibles par le fichier Portal.jsp, les différentes fonctionnalités du back-office sont accessibles par leurs propres fichiers ".jsp", par exemple:

- *<webapp>/jsp/admin/ManageProperties.jsp*
- *<webapp>/jsp/admin/system/ManagePlugins.jsp*

L'accès au Back Office nécéssite une authentification. Les liens vers les différents contenus sont ajoutés dynamiquement dans la page d'accueil en fonction des droits et des roles de l'utilisateur back office.

Note: Nous parlons ici du méchanisme d'authentification Back Office. Il existe également un mécanisme d'authentification Front Office optionnel (plugin mylutece). Ces deux mécanismes ne doivent pas être confondus: ils sont indépendants et les deux populations d'utilisateurs totalement séparées.

TERMINOLOGIE: LUTECE-CORE / LUTECE-PLUGIN

L'architecture modulaire de Lutece s'appuie sur la distinction entre le coeur et les plugins.

Le **coeur** fournit un framework avec lequel les plugins apportent des fonctionnalités.
Les avantages sont de permettre une homogénéisation tant au niveau de l'implémentation que de l'interface utilisateur, sans surcharger de complexité tous les cas d'utilisation.
Les principes de développement et d'organisation du code sont partagés entre le coeur et les plugins. Par exemple, le coeur et le plugin-document implémentent tous les deux des Content Services selon la même architecture.
Dans la suite de ce livre, nous décrirons tantôt des fonctionnalités du coeur, tantôt des fonctionnalités de plugins, car les principes restent globalement les mêmes.

Dans la terminologie Lutece, en plus du coeur et des plugins, il existe aussi un autre type de composant: les **modules**.
Leur role est d'implémenter ou de spécialiser des fonctionnalités d'un plugin. Il s'agit d'une convention de nommage, car leur comportement est identique à celui d'un plugin. Un exemple d'implémentation de ce type de composant est le module mylutece-database dont le role est d'implémenter les fonctionnalités d'authentification du plugin-mylutece en se basant sur des utilisateurs stockés dans la base de donnée locale. Dans le cas ou la webapp Lutece doit etre protégée par une authentification de type SSO, partagée entre plusieurs sites, ce module peut etre substitué par un module d'authentification SSO, tel que le module mylutece-cas ou mylutece-openam qui interagissent avec des serveurs distants d'authentification.

2. DESCRIPTION FONCTIONNELLE DES COMPOSANTS

Dans ce chapitre, nous donnerons quelques exemples de composants Lutece, vus sous l'angle fonctionnel, avant de préciser plus loin la façon dont ils composent les plugins et comment les développer.

Voici donc quelques composants standards dans la terminologie Lutece.

LES PORTLETS

Les pages de Lutèce sont composées de portlets. Ces derniers sont des blocs de contenus positionnés dans une page du site.

Ils sont essentiellement utilisés pour la partie éditoriale.

LES XPAGES

Les Pages Spéciales (XPage) sont des composants qui permettent de publier en *Front Office* des pages qui n'appartiennent pas à l'arborescence éditoriale du CMS (pages composées avec les portlets ci-dessus).

Elles sont destinées à proposer des contenus de type applicatif. Exemples : wiki (ci-dessous), formulaire de contact, moteur de recherche, plan du site, ...

L'url d'accès à ces pages se fait en utilisant le paramètre page=<*xpage-name*>. Ex : `/jsp/site/Portal.jsp?page=wiki`

LES ADMIN FEATURES

Les fonctionnalités d'administration correspondent à l'ensemble des fonctions disponibles dans le menu du Back Office.

Pour chaque fonctionnalité correspond un droit qui peut être attribué à un utilisateur. Lorsque l'utilisateur est autorisé, la fonctionnalité s'affiche dans le menu du *back office* **Lutece**.

Contenu ▾ Site ▾ Applications ▾ Utilisateurs ▾ Gestionnaires ▾ Charte ▾ Système ▾ admin Admin ▾

Informations Système

Informations Système

Lancer le Garbage Collector

Version du Java Runtime Environment	1.7.0_60
Auteur du Java Runtime Environment	Oracle Corporation
Version des spécifications de la JVM	1.7
Auteur des spécifications de la JVM	Oracle Corporation
Nom des spécifications de la JVM	Java Virtual Machine Specification
Version de l'implémentation de la JVM	24.60-b09
Auteur de l'implémentation de la JVM	Oracle Corporation
Nom de l'implémentation de la JVM	Java HotSpot(TM) 64-Bit Server VM
defaultCharset de la JVM	US-ASCII
Version des spécifications du JRE	1.7
Auteur des spécifications du JRE	Oracle Corporation
Nom des spécifications du JRE	Java Platform API Specification
Nom du Système d'exploitation	Linux
Version du Système d'exploitation	2.6.32-431.el6.x86_64
Infos du Servlet Container	Apache Tomcat/6.0.41

Dernier redémarrage de la webapp : Aug 13, 2015 1:32:49 PM

Informations sur les pools de connexions aux bases de données

Pool	Nombre de connexions ouvertes / Max.
portal	- / - (Tomcat)

Informations Mémoire

Mémoire totale dans la JVM	377,50 Mo
Mémoire libre dans la JVM	283,80 Mo
Mémoire maximum dans la JVM	910,50 Mo

LES DAEMONS

Toute sorte de traitements automatiques peuvent être lancés en arrière-plan de manière asynchrone à travers des `Daemons` applicatifs : envoi de mails, indexation, purge de données, mise à jour du cache.

LUTECE Contenu ▾ Site ▾ Applications ▾ Utilisateurs ▾ Gestionnaires ▾ Charte ▾ Système ▾ admin Admin

Gestion des Daemons

Liste des daemons

Id	Nom - description	Démarré	Intervalle(s)	Dernière exécution	Actions
seoFriendlyUrlGenerator (seo)	Démon de génération des URL explicites Genère des URL explicites pour toutes les nouvelles ressources	Activé	3600	11/12/2013 17:36 - Logs seoFriendlyUrlGenerator - seo...	
databaseAccountLifeTimeDaemon (mylutece-database)	Daemon de durée de vie des comptes utilisateurs Daemon de gestion des durées de vie des comptes utilisateurs	Activé	86400	11/12/2013 05:36 - Logs databaseAccountLifeTimeDaemon - mylutece-database...	
seoSitemapGenerator (seo)	Démon de génération du Sitemap Génère le Sitemap à partir des URLs explicites (si activé dans les Optimisations SEO)	Activé	3600	11/12/2013 17:36 - Logs seoSitemapGenerator - seo...	
anonymizationDaemon (core)	Daemon d'anonymisation des administrateurs Anonymise les administrateurs de l'application lorsque leur compte a expiré	Activé	86400	11/12/2013 05:36 - Logs anonymizationDaemon - core...	
directoryWorkflowRecordRemovalDaemon (directory)	Daemon de suppression des entrées des annuaires en fonction de leur etat de workflow Supprime les fiches des annuaires ayant un etat défini dans les paramètres avancés de l'annuaire	Activé	3600	11/12/2013 17:36 - Logs directoryWorkflowRecordRemovalDaemon - directory...	
threadLauncherDaemon (core)	Daemon de gestion de threads Daemon permettant d'executer des actions dans des threads	Activé	60	11/12/2013 18:26 - Logs threadLauncherDaemon - core...	
indexer (core)	Indexation du moteur de recherche Indexe les contenus pour le moteur de recherche	Activé	300	11/12/2013 18:23 - Logs indexer - core...	
resetRssFeeds (rss)	Nettoyage du cache RSS Supprime la mémoire cache pour recharger le fichier de cache	Activé	300	11/12/2013 18:26 - Logs resetRssFeeds - rss...	
updaterStatus (updater)	UpdateStatus daemon Recherche les mises à jour sur les dépôts	Activé	300	11/12/2013 18:22 - Logs updaterStatus - updater...	
formRemoveResponses (form)	Daemon de suppression des réponses Daemon de suppression des réponses	Désactivé	86400	- Logs formRemoveResponses - form...	

LES LINKSERVICES

Il est possible d'insérer via un bouton placé dans l'éditeur riche du *Back Office* des ressources **Lutece** sous forme d'inclusions HTML

LUTECE Contenu Site Applications Utilisateurs Gestionnaires Charte Système admin Admin

Administration du Site

Page accueil

Modifier une rubrique Html ou Texte libre

Identifiant de la page

1

Colonne

2

Ordre

1

Style

Défaut

Restriction au role

No restriction

Accepte Alias

Oui Non

Afficher sur des écrans de type

Téléphone

Titre de la rubrique *

Bienvenue sur la démo

Affichage

avec titre sans titre

Contenu HTML :

Accès à l'administration du

Choisissez votre profil :

Administrateur technique

Webmaster - Gestion de cont

Rédacteur - Producteur de co

Se connecter

Service d'insertion de code HTML de Lutece - Mozilla Firefox

dev.lutece.paris.fr/site-demo/jsp/admin/insert/GetAvailableInsertSer

Retour

INSÉRER

- Un média de la base des documents (image, fichier, vidéo, ...)
- Un lien vers un document
- Sélection de documents depuis la newsletter
- Un média de votre ordinateur (image, fichier, vidéo, ...)
- Une vidéo de YouTube/Google

Rechercher

3. DESCRIPTION DE L'ARCHITECTURE LOGICIELLE

Ce chapitre a pour objectif de décrire l'architecture logicielle de **Lutece**. Nous allons commencer par la description des différentes étapes de la requète HTTP d'un client à la réponse HTTP du serveur. Puis nous décrirons plus en détail l'organisation du code dans les différentes classes java, JSP, fichiers HTML et xsl. Enfin, nous verrons de manière exhaustive l'organisation de ces fichiers dans un projet lutece.

ARCHITECTURE GÉNÉRALE

Le choix général d'architecture repose sur la spécification Java EE qui apporte un cadre aux applications d'entreprise n-tiers.

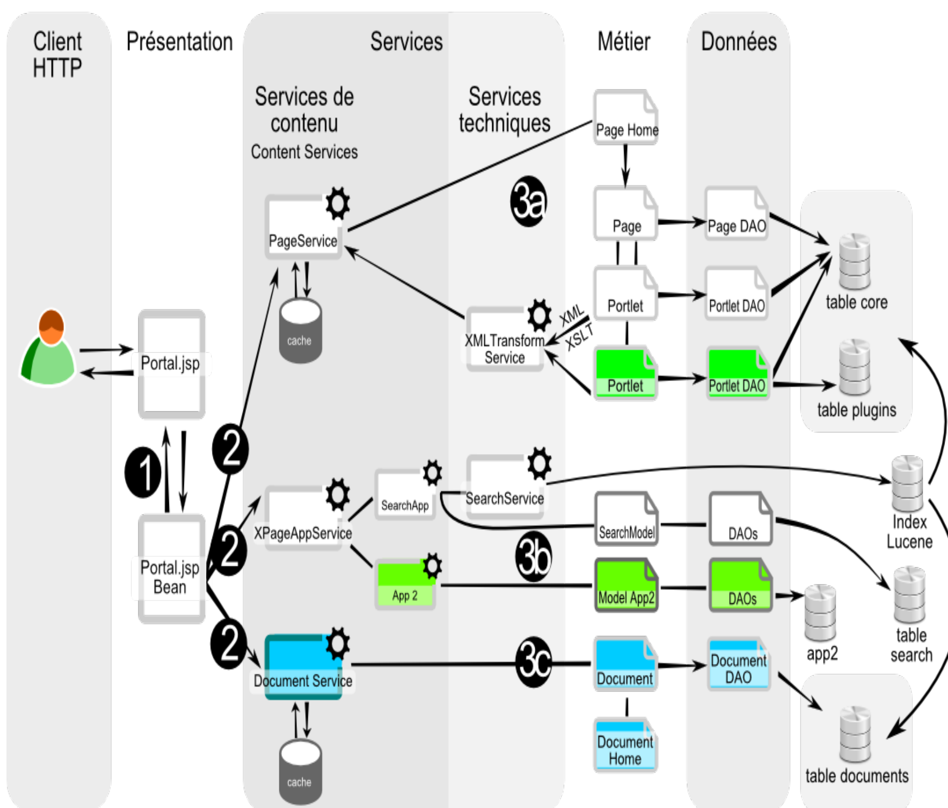
DÉCOUPAGE GÉNÉRAL DE L'ARCHITECTURE

Les choix de la conception détaillée se sont portés sur une architecture en quatre couches côté serveur :

- La couche présentation assurant les fonctions de gestion de l'interface utilisateur.
- La couche service assurant les fonctions d'interface entre la couche métier et la couche présentation, ainsi qu'un ensemble de services techniques : cache, transformations XML/XSL vers HTML ou HTML vers XHTML.
- La couche métier assurant la gestion des objets modélisant l'activité métier.
- La couche d'accès aux données qui assure la persistance des objets métiers.

SCHÉMA DE RESTITUTION DU CONTENU

Le schéma suivant présente de manière simplifiée le processus de restitution d'un contenu Front Office. Le schema se lit de gauche à droite et représente les différentes étapes commençant à la requète HTTP du client et terminant à la réponse HTTP du serveur. Le schéma est suivi d'explications sur les différentes étapes, puis de descriptions détaillées des différentes couches.



Les différents composants sont representés par des symboles différents: ceux de la couche présentation par de simples rectangles, ceux de la couche service avec un engrenage, ceux de la couche métier et ceux de la couche d'accès au données par une feuille de papier.
Les couleurs indiquent en blanc les composants de lutece-core, en bleu les composants du plugin-document, un plugin standard de lutèce, et en vert des composants qui pourraient être apportés par un nouveau plugin.
Ce schema donne donc plusieurs exemples d'architecture dont on peut s'inspirer lors de l'écriture de nouveaux plugins.

Les étapes de restitution

1. Comme vu précédemment, `Portal.jsp` est le point d'entrée de restitution.
   Comme toutes les JSPs dans Lutece, `Portal.jsp` utilise une directive <jsp:useBean> pour charger un bean java (ici `PortalJspBean`), ce qui permet d'écrire le code dans une classe java normale afin de faciliter la maintenance. `PortalJspBean` implémente des methodes correspondant à différentes opérations de l'utilisateurs, et délègue la plupart des fonctionnalités à des classes java de services : `PortalService`, `ContentService`, `AppPropertiesService`, etc. Il n'est pour l'instant pas important de connaitre tous ces services, et les plus importants seront présentés dans la suite de livre, mais l'organisation du code JSP/JSPBean/Service est un point central de Lutece.

2. La suite du traitement consiste à utiliser les paramètres HTTP pour sélectionner un `Content Service`.
   Les Content Services sont des classes java comprenant principalement une méthode `isInvoked` indiquant selon les paramètres HTTP si c'est ce service qui est demandé, et une methode `getPage` renvoyant le contenu HTML en utilisant un cache. Les Contents Services proviennent du coeur de Lutece ou de plugins.
   Pour reprendre les exemples du précédant chapitre, voici différents `Content Services` :
   1. `PageContentService` de `lutece-core`, paramètres `HTTP "page_id=n"`. Ce `Content Service` gère les pages éditoriales du site. Ces pages sont configurées en *back office* en y organisant des portlets. Ces pages sont organisée en une arborescence du site Lutece reflétée dans les menus et le fil d'arianne.
   2. `XPageAppService` de `lutece-core`, paramètres `HTTP "page=name"`. Ce `Content Service` gère des pages autonomes appelée `XPages` fournissant des fonctionnalités spécifiques. Le contenu est entièrement fabriqué par la classe java appelée et n'est généralement pas aussi éditable que les pages du `PageContentService` avec leurs portlets.
   3. `DocumentContentSerivce` de plugin-document, paramètres `HTTP "document_id=n&porlet_id=m"`. Ce Content service est fourni par le plugin document qui propose dans lutece des contenus typés (par exemple, article, brève, etc.) organisés autour d'un workflow de publication (gestion de la collaboration, de la validation).

3. 3a) Si une page du `PageContentService` est demandée, le bean s'adresse au `PageService` qui fournira la page si elle est dans son cache ou sinon il la construira. La construction d'une page fait appel à la classe `PageHome` pour obtenir une instance de la page demandée. Ensuite, le contenu de chaque rubrique de la page est récupéré en XML puis transformé à la l'aide de la feuille de style XSL associée, et agrégé pour constituer la page. La nouvelle page construite est alors mise dans le cache. Ici, le point important d'architecture est que les classes services s'appuient sur des classes d'objets métiers, par exemple `PageHome` et `Page`, qui chargent leur données depuis des objets DAO (*Data Acess Object*), par exemple `PageDAO`.
   3b) et 3c) fonctionnent sur le même principe en s'appuyant sur des services techniques et des objets metiers différents.

ARCHITECTURE LOGICIELLE PAR BRIQUE : DESCRIPTION DES COUCHES LOGICIELLES

COUCHE DE PRÉSENTATION

Cette couche est réalisée à l'aide de JSP, de Java Beans, éventuellement de servlets. Le code HTML est produit de deux manières :

- Pour les éléments les plus dynamiques du portail (par exemple les documents de plugin-document), le HTML est produit par transformation du contenu XML à l'aide de feuilles de style. Le contenu XML pourra éventuellement contenir lui-même du code HTML pour tout ce qui concerne les contenus non structurés et notamment les pages locales.

- Pour les élements qui ne requierent pas le support de plusieurs formats de restitution (par exemple le module d'administration ou les `XPages`), des fichiers contenant des modèles de représentation HTML appelés `Templates` seront utilisés.

LES JSP

Pour rappel, nous parlons ici de la catégorie à laquelle appartient Portal.jsp, le point d'entrée principal de restitution de contenus du *Front Office*.

De manière générale, le rôle attribué aux JSP est strictement limité à la gestion de la structure des pages et à la navigation. La partie HTML sera construite par des beans qui utiliseront des templates pour construire le code qui sera inséré dans les pages JSP.

Les JSP contiendront donc essentiellement :

- la directive `<jsp:useBean>` qui charge un bean contenant la majorité du code.
- la directive de redirection sur la page d'erreur : `<%@ page errorPage="path/to/ErrorPage.jsp"%>`
- des directives d'inclusion d'objets statiques :`<jsp:include>` (ex : code HTML pour inclure une feuille de style)
- des branchements conditionnels (`<jsp:forward>`)

Tout code Java et HTML est proscrit dans ces pages.

Les JSP doivent toutes contenir une JSP d'entête et une JSP de fin de page. Ceci permet d'intégrer dans toutes les pages des traitements initiaux et finaux sans avoir à modifier l'ensemble des pages (ex : test du timeout de la session, entête et pied de page HTML, lien vers une feuille de style, ajout de traitements statistiques, ...).

Le Front Office n'utilise principalement que Portal.jsp, alors que le Back Office utilise de nombreuses JSP. Cela permet d'avoir plus d'homogénéité dans le Front Office et plus de flexibilité dans le Back Office.

LES JSPBEANS

Pour rappel, nous parlons ici de la catégorie à laquelle appartient PortalJspBean, le bean associé à la JSP Portal.jsp vue dans les exemples.

Ces beans sont chargés de gérer la présentation des informations issues des objets métier. Ils doivent notamment produire du code HTML qui sera inséré dans les JSP. Ces composants sont regroupés dans le package **fr.paris.lutece.portal.web** pour le coeur de lutece, **fr.paris.lutece.plugins.*.web** pour les plugins. et ont un suffixe *JspBean* pour les différencier des beans métier.

Les JSP beans peuvent être utilisés par plusieurs pages JSP traitant des mêmes informations, notamment pour partager des constantes.

La portée des JSP beans (attribut `scope` dans la déclaration du bean dans la JSP) doit être soit session s'il contient des variables d'instance relative à la session de l'utilisateur (état conversationnel), soit request s'il n'a pas de variable d'instance. Dans certains cas, les scopes `page` ou `applications` sont aussi utilisés.

LES TEMPLATES

Les templates sont des fichiers .html contenant un template de code HTML et éventuellement Javascript. Le moteur de template utilisé est FreeMarker *http://freemarker.org*. Par exemple, le template principal associé à Portal.jsp est le fichier `page_frameset.html`.

Après le templating Freemarker, certaines transformations additionelles sont effectuées dans cet ordre:

1. Le traitement des libellés multilingue introduits sous la forme `#i18n{key}`. La gestion de l'internationalisation (i18n) décrite plus loin dans un chapitre dédié

2. Le traitements des libellés sous la forme `#dskey{key}` dont les valeurs sont sockées en base de données (dans le datastore de Lutece).

3. Les traitements d'éventuels `ContentPostProcessor` comme ceux des plugins plugin-extend ou plugin-seo.

L'avantage de ces templates est qu'ils peuvent être partagés par plusieurs JSP et qu'ils peuvent être modifiés par des concepteurs graphiques sans risque d'impact sur les traitements. Ces templates seront stockés dans le répertoire */WEB-INF/templates*.

LES FEUILLES DE STYLE XSL

Les feuilles de style XSL servent à transformer les contenus XML en HTML. Par ce mécanisme, on sépare le contenu de la forme. Par simple changement de style les mêmes contenus peuvent être présentés de manière très différentes.

LA COUCHE MÉTIER

La couche métier est réalisée par un ensemble de classes correspondant à des objets métiers.

Ces classes ne contienent aucun code technique HTML ou SQL. Ces composants sont regroupés dans les packages **fr.paris.lutece.portal.business.*** pour le coeur et **fr.paris.lutece.plugins.*.business.*** pour les plugins.

La persistance de ces objets est assurée par des objets DAO (`Data Access Object`) dont l'interface contient les principales méthodes d'accès aux données correspondant aux opérations SQL de base : `load` (`SELECT`), `store` (`UPDATE`), `create` (`INSERT`), `delete` (`DELETE`).

Nous avons décidé d'avoir une conception proche des Enterprise JavaBeans, et nous utilisons une classe "Home" pour chaque objet métier principal c'est à dire disposant d'un DAO, inspirée des EJB Home Interface. Les prérogatives de ces classes sont, comme pour les EJB, d'assurer la gestion des instances de l'objet métier :

- Création d'une instance (méthode `create()`)
- Renvoi d'une instance à partir de sa clé primaire (méthode `findByPrimaryKey()`)
- Renvoi d'une collection d'instances pour un critère donné (méthodes `findByCritère()`)

À ces méthodes qui figurent dans l'interface des classes Home des EJB entity, nous ajouterons les méthodes qui normalement correspondent à des actions gérées par le conteneur d'EJB. L'objet Home étant le seul objet ayant accès au DAO, c'est lui qui assurera les opérations de modification et de suppression des entités dans la base de données.

Les appels à ces méthodes seront effectués par le biais de méthodes `update()` et `remove()` au niveau de l'objet métier.

Voici le tableau récapitulatif des interfaces classiques des différents objets. Une ligne represente des méthodes liées, leur portée, et l'existance de cette méthode dans le modèle de programmation des EJB entity :

Objet Métier		Home			DAO		
Méthode	**Portée**	**Méthode**	**Portée**	**EJB?**	**Méthode**	**Portée**	**EJB?**
		`create`	*public*	X	`insert`	*package*	X
		`findByPrimaryKey`	*public*	X	`load`	*package*	X
		`finbBy...`	*public*	X	`selectBy`	*package*	X
Update	*public*	`update`	package		`store`	*package*	X
Remove	*public*	`remove`	package		`delete`	*package*	X

Une recommandation importante de la conception d'EJB entity est d'avoir une granularité assez grosse, c'est à dire que tout objet métier ne doit pas nécessairement être implémenté sous la forme d'un EJB entity. Tous les objets métiers dépendant d'un objet métier principal, et particulièrement dans le cas où il est en lecture seule, doivent être implémentés sous la forme d'une classe simple. La lecture de cet objet sera alors réalisée au niveau du DAO de l'objet principal.

LES SERVICES

Les services servent à implémenter des fonctionnalités qui ne sont pas directement reliées à la création et modification d'un objet métier et de ses données. C'est souvent la partie la plus importante en termes de répartition de fonctionnalités d'une application.

C'est dans cette catégorie que se trouvent les Content Services, qui implémentent le framework dans lequel la majorité des contenus du Front Office sont restitués: PageContentService et XPageAppService.

Un certain nombre de services techniques fournis par le coeur sont accessibles à partir des composants de l'application : Par exemple:

Service	Description
AppTemplateService	Renvoie un modèle HTML à partir de son nom. Ce service dispose d'un cache activable qui permet de ne pas relire sur disque un modèle qui a déjà été demandé.
AppConnectionService	Fourniture de connexions à la base données à partir d'un pool. Ce service est paramétrable pour utiliser soit le pool du serveur d'application, soit un pool de connexions autonome.
AppPropertiesService	Renvoie les valeurs des variables définies dans les fichiers properties des répertoires /WEB-INF/conf/ et /WEB-INF/conf/plugins/
MailService.java	Permet l'envoi de mail en format text ou html, avec gestion d'évenements de calendrier.
I18nService.java	Gère la traduction de chaines de caractères en fonction du choix de l'utilisateur
AppLogService.java	Gère les logs avec différents niveaux de sévérité, différents loggers
XmlTransformerService.java	Gère l'application de feuilles de styles XSL pour transformer des contenus XML.

Il ne s'agit pas ici de donner une liste exhaustive des différents services du coeur, car la nature des services fait que des fonctionnalités très variées sont y sont implémentées. Il est conseillé de parcourir le code du coeur et de plugins incontournables (par exemple plugin-document, plugin-directory) pour avoir des exemples de fonctionnalités à regrouper dans des classes de services.

BASE DE DONNÉES LUTÈCE

Les requêtes SQL écrites dans les fichiers DAO doivent être compatible avec la norme SQL92.

Les requêtes SQL et les types de colonnes dans les scripts de création de la base de données doivent être compatibles avec les bases de données MySQL ou MariaDB. Le système de Lutece de création de la base de données basé sur **Ant** assure une transcription pour les SGDB PostgreSQL, HSQLDB et Oracle.

ORGANISATION PHYSIQUE DES FICHIERS DE L'APPLICATION

L’arborescence des répertoires dans le projet source de Lutece core est définie comme suit et intègre les spécifications JEE - Servlet 2.5.

ORGANISATION DES RÉPERTOIRES DU NOYAU

Localisation	Type de fichier	Description
src/java	*.java	Les sources de l’application
src/sql	*.sql	Les scripts de création de base et d’initialisation de l’application
<webapp>/css	*.css	Feuille de style CSS par défaut utilisée par l’application
<webapp>/images	*.gif,*.jpg,*.png	L'ensemble des pictogrammes utilisés par l’application
<webapp>/js	*.js	Fichiers javascripts utilisés dans l’ensemble de l’application
<webapp>/jsp/admin	*.jsp	Les Java Server Pages du module d’administration de l’application
<webapp>/jsp/admin/plugins	*.jsp	Les Java Server Pages de la fonctionnalité de gestion des plugins du module d’administration
<webapp>/jsp/site	*.jsp	Les Java Server Pages de la partie site de l’application
<webapp>/WEB-INF	web.xml	Fichier de configuration de l’application web (specs Servlet 2.5). Contient la déclaration des servlets de l’application

<webapp>/WEB-INF/conf	*.properties	• config.properties : le fichier de configuration de l'application.
		• db.properties : le fichier de configuration des pools de connexions aux bases de données. Il comprend notamment les déclarations suivantes :
		  ◦ Drivers JDBC
		  ◦ Sources de données
		  ◦ Nombre de connexions ouvertes à la création d’un pool
		  ◦ Nombre maximum de connexions par pool
		Ce fichier est utilisé pour un fonctionnement autonome (sans serveur d’application).
		• Autres fichiers properties

<webapp>/WEB-INF/index	_*.f* et autres	Les index du moteur de recherche Lucene pour le site de l’application
<webapp>/WEB-INF/indexall	_*.f* et autres	Les index du moteur de recherche Lucene pour un ensemble de sites définis dans le fichier config.properties
<webapp>/WEB-INF/lib	*.jar	Les fichiers archive contenant les classes importées dans l’application ainsi que ceux de lutece core. (specs Servlet 2.5)
<webapp>/WEB-INF/logs	*.logs	Les logs de l’application.
<webapp>/WEB-INF/taglibs	*.tld	Fichiers taglibs utilisés dans l’application
<webapp>/WEB-INF/templates/admin et sous répertoires	*.html	Les modèles HTML utilisés pour la construction dynamique des pages. Il s’agit de blocs de code HTML utilisés par les beans de présentation du module d’administration
<webapp>/WEB-INF/templates/site et sous-répertoires	*.html	Les modèles HTML utilisés pour la construction dynamique des pages. Il s’agit de blocs de code HTML utilisés par les beans de présentation du site
<webapp>/WEB-INF/tmp	aucun fichier	Répertoire utilisé dans les fonctionnalités d’ upload , doit être toujours vidé après le traitement
<webapp>/WEB-INF/xsl/admin	*.xsl	Les feuilles de style XSL de mise en forme du contenu XML sur le module d'administration
<webapp>/WEB-INF/xsl/normal	*.xsl	Les feuilles de style XSL de mise en forme du contenu XML sur le site

ORGANISATION DES RÉPERTOIRES DES PLUGINS

Un plugin peut nécessiter un ensemble assez important et divers de fichiers. Voici les répertoires désignés pour contenir ces fichiers.

Localisation	Type de fichier	Description
/src/java/fr/lutece/plugins/<plugin_name>/business	*.java	Les fichiers sources java de la couche métier
/src/java/fr/lutece/plugins/<plugin_name>/service	*.java	Les fichiers sources java de la couche service
/src/java/fr/lutece/plugins/<plugin_name>/web	*.java	Les fichiers sources java de la couche présentation
/src/sql/plugins/<plugin_name>	*.sql	Les scripts SQL d'installation et d'initialisation des tables du plugin
<webapp>/jsp/admin/plugins/<plugin_name>	*.jsp	Les JSP des fonctions d'administration

<webapp>/images/local/skin/plugins/<plugin_name>	*.gif,*.jpg,*.png	Les images des fonctions d'administration
<webapp>/images/local/skin/plugins/<plugin_name>	*.gif,*.jpg,*.png	Les images de présentation de l'application
<webapp>/images/local/data/<plugin_name>	*.gif,*.jpg,*.png	Les images gérées comme des données du plugin
<webapp>/plugins/<plugin_name>/*.*	Sous-répertoires, tous types de fichiers	Emplacement réservé aux plugins ayant besoins de répertoires ou fichiers spécifiques
<webapp>/WEB-INF/conf/plugins/<plugin_name>.properties	*.properties	Le fichier de propriété .properties du plugin
<webapp>/WEB-INF/plugins	*.xml, plugins.dat, plugin_2_2.dtd	Le fichier de définition du plugin
/WEB-INF/templates/admin/plugins	*.html	Les templates des fonctions d'administration
/WEB-INF/templates/skin/plugins	*.html	Les templates de l'application accessibles du portail
/WEB-INF/lib/plugin_<plugin_name>_<version>.jar	*.jar	Le fichier jar contenant les classes du plugin

4. DESCRIPTION DE LA COMPOSITION DES PLUGINS

Les plugins sont des macro-composants que l'on peut ajouter à Lutèce pour étendre ses fonctionnalités. Ils ont pour vocation d'intégrer différents types de fonctionnalités en respectant un mode d'installation standard.

LES COMPOSANTS APPORTÉS PAR LES PLUGINS

Les composants pouvant être ajoutés par les plugins sont les suivants :

- 0 à n feuilles de style CSS spécifiques
- 0 à n scripts Javascript spécifiques
- 0 à n fonctions d'administration
- 0 à n portlets
- 0 à 1 service de contenu
- 0 à 1 services d'insertion de type LinkService
- 0 à 1 services d'insertion de type HtmlService
- 0 à n servlets, filtres de servlet o listeners
- 0 à n tableau de bord (dashboard)
- 0 à n daemons
- 0 à n indexeurs de recherche
- 0 à n macro freemarker

LES PLUGINS ET LES BASES DE DONNÉES

Un plugin nécessitant le stockage de données accède aux informations contenues dans son modèle de données à travers un pool de connexions.

Par défaut le pool de connexions utilisé est celui du coeur, mais il est possible d'en déclarer un spécifiquement pour un plugin.

Cette déclaration se fait dans le fichier `db.properties` présent dans le répertoire`/WEB-INF/conf/` .

L'association entre le plugin et le pool de connexions est initialisée à partir du `plugin.dat` et peut par la suite être modifiée dans le back office de l'application.

CHARGEMENT DES PLUGINS

Les plugins sont chargés au démarrage de la webapp par le service `PluginService` qui recherche tous les fichiers XML se trouvant dans le répertoire `/WEB-INF/plugins` .

Les informations sur le statut local des plugins (état installé/non-installé, pool de connection spécifique, ...) sont initialisées à partir du fichier `plugins.dat` situé dans le même répertoire.Lorsque ces informations sont mises à jour en back office, les modifications apportées sont persistées en base de données.

Si un plugin dispose d'un fichier properties, celui doit se trouver dans le répertoire `/WEB-INF/conf/plugins`. Les propriétés de ce plugin seront automatiquement chargées et ajoutées à l'ensemble des propriétés de l'application. Elles seront alors accessibles par le biais de la classe `AppPropertiesService` . Toutes les propriétés d'un plugin doivent être préfixées par le nom du plugin pour éviter tout conflit de clés.

Important : Un mécanisme de surcharge (override) est disponible depuis la version 4 de Lutece. Il est possible de surchager une partie ou la totalité des valeurs d'un fichier properties en plaçant ces valeurs dans un fichier situé dans le répertoire `/WEB-INF/conf/override/plugins`

L'ARBORESCENCE DE FICHIERS DES PLUGINS

Un plugin peut contenir un ensemble assez important et divers de fichiers. C'est pourquoi il est n'écessaire de les organiser de manière arborescente.

Type de fichier	Répertoire
Les JSP des fonctions d'administration	/jsp/admin/plugins/<plugin_name>/*.jsp
Les JSP accessibles du portail	/jsp/site/plugins/<plugin_name>/*.jsp
Les images des fonctions d'administration	/images/admin/skin/plugins/<plugin_name>/*.*
Les images de présentation de l'application	/images/local/skin/<plugin_name>/*.*
Les images gérées comme des données du plugin	/images/local/data/<plugin_name>/*.*
Le fichier properties du plugin	/WEB-INF/conf/plugins/<plugin_name>.properties
Le fichier de définition du plugin	/WEB-INF/plugins/<plugin_name>.xml
Les templates des fonctions d'administration	/WEB-INF/templates/admin/plugins/<plugin_name>.properties
Les templates de l'application accessibles en front office	/WEB-INF/templates/skin/plugins/<plugin_name>.properties
Le fichier jar contenant les classes du plugin(fichier généré lors du build)	/WEB-INF/lib/plugin_<plugin_name>_<version>.jar
Les scripts SQL d'installation et d'initialisation des tables du plugin (distribution binaires)	/WEB-INF/sql/plugins/<plugin_name>/*.sql

L'organisation des fichiers sources est par ailleurs la suivante

Type de fichier	Répertoire
Les fichiers sources java de la couche métier	/src/java/fr/lutece/plugins/<plugin_name>/business/*.java
Les fichiers sources java de la couche service	/src/java/fr/lutece/plugins/<plugin_name>/service/*.java
Les fichiers sources java de la couche présentation	/src/java/fr/lutece/plugins/<plugin_name>/web/*.java
Les ressources de type messages dans les différentes langues	/src/java/fr/lutece/plugins/<plugin_name>/resources/*.properties
Les scripts SQL d'installation et d'initialisation des tables du plugin (distribution sources)	/src/sql/plugins/<plugin_name>/*.sql
Les documentations au format XML pour Maven	/src/site/xdoc/[fr/]/xdoc/plugins/<plugin_name>/*.xml

LE FICHIER DE CONFIGURATION DU PLUGIN

Le fichier de configuration d'un plugin est un fichier XML se présentant sous la forme suivante :

```
<?xml version=1.0 encoding=ISO-8859-1?>
<plug-in>
      <name>myplugin</name>
      <class>fr.paris.lutece.portal.service.PluginDefaultImplementation</class>
      <version>1.0</version>
      <description>Description of the plugin</description>
      <provider>Mairie de Paris</provider>
      <provider-url>http://lutece.paris.fr</provider-url>
      <icon-url>../../images/admin/skin/plugins/myplugin/myplugin.gif</icon-url>
      <copyright>Copyright (c) 2001-2015 Mairie de Paris</copyright>

      <!-- Le plugin requiert-il un pool de connexions : 1 - oui, 0 - non -->
      <db-pool-required>1</db-pool-required>

      <!-- Feuilles de style CSS spécifiques éventuelles -->
      <css-stylesheets>
      <css-stylesheet>myplugin/myplugin.css</css-stylesheet>
      </css-stylesheets>

      <!-- Scripts eventuels -->
      <javascript-files>
             <javascript-file>myplugin/myplugin.css</javascript-file>
      </javascript-files>

      <!-- Listes des fonctionnalités d'administration éventuellement introduites par le plugin-->
      <admin-features>
             <admin-feature>
                    <feature-id>MYFEATURE_MANAGEMENT</feature-id>
                    <feature-title>My Feature</feature-title>
                    <feature-description>Description of my feature</feature-description>
                    <feature-level>3</feature-level>
                    <feature-url>plugins/myplugin/ManageMyFeature.jsp</feature-url>
             </admin-feature>
...
      </admin-features>

      <!-- Listes des portlets éventuellement introduits par le plugin-->
      <portlets>
             <portlet>
                    <portlet-class>fr.paris.lutece.plugins.myportlet.business.portlet.MyPortletHome</portlet-class>
                    <portlet-type-name>MyNew Portlet</portlet-type-name>
                    <portlet-creation-url>plugins/article/CreatePortletMyPortlet.jsp</portlet-creation-url>
                    <portlet-update-url>plugins/article/ModifyPortletMyPortlet.jsp</portlet-update-url>
             </portlet>
...
      </portlets>

      <!-- Application à base de pages spéciales introduite éventuellement par le plugin -->
      <applications>
             <application>
                    <application-id>app_page_name</application-id>
                    <application-class>fr.paris.lutece.plugins.myplugin.web.MyPluginApp</application-class>
                    <application-roles>role1,role2</application-roles>
             </application>
      </applications>

      <!-- Content Service -->
      <content-service>
             <content-service-class>fr.paris.lutece.plugins.myplugin.service.MyContentService</content-service-class>
      </content-service>

      <!-- Links Service -->
             <link-service>
                    <link-service-id>mylinkservice</link-service-id>
                    <link-service-class>fr.paris.lutece.plugins.mylinkservice.service.MyLinkService</link-service-
class>
                    <link-service-bean-class>fr.paris.lutece.plugins.mylinkservice.web.MyLinkServiceJspBean</link-
service-bean-class>
                    <link-service-label>Link to my URIs</link-service-label>
             </link-service>

      <!-- Servlets-->
      <servlets>
             <servlet>
                    <servlet-name>myServlet1</servlet-name>
                    <url-pattern>/servlet/plugins/myplugin/myServlet1</url-pattern>
                    <servlet-class>fr.paris.lutece.plugins.myplugin.web.MyFirstServlet</servlet-class>
             </servlet>
             <servlet>
                    <servlet-name>myServlet2</servlet-name>
                    <url-pattern>/servlet/plugins/myplugin/myServlet2</url-pattern>
                    <servlet-class>fr.paris.lutece.plugins.myplugin.web.MySecondServlet</servlet-class>
             </servlet>
      </servlets>

      <!-- filters -->
      <filters>
             <filter>
                    <filter-name>myFilter</filter-name>
                    <url-pattern>/*</url-pattern>
                    <filter-class>fr.paris.lutece.plugins.myplugin.web.MyFilter</filter-class>
                    <init-param>
                           <param-name>param1</param-name>
                           <param-value>value of param1</param-value>
                    </init-param>
             </filter>
      </filters>

</plug-in>
```

Une DTD des plugins est disponible dans le répertoire /WEB-INF/plugins de la Webapp de Lutece.

MODULES

Certains plugins peuvent avoir leurs propres plugins. Ceux-ci sont appelés modules. Les plugins tels que mylutece, formengine, codewizard disposent de modules.

Les règles concernant les modules sont les suivantes :

Type de fichier	Répertoire
Packages des fichiers sources java	/src/java/fr/lutece/plugins/<plugin_name>/modules/<module_name>//*.java
Localisation des fichiers templates HTML	/plugins/<plugin_name>/modules/<module_name>//*.html
Nom et emplacement du fichier XML du plugin	/WEB-INF/plugins/<plugin_name>-<module_name>.xml
Nom et emplacement du fichier de configuration	/WEB-INF/conf/plugins/<plugin_name>-<module_name>.properties

AJOUTER UN PLUGIN À UN SITE EXISTANT

Il est possible d'ajouter un plugin à un site existant sans utiliser le système d'assemblage de Lutece décrit plus loin dans le livre. Contrairement à un site utilisant ce système, cette méthode présente le gros inconvenient de ne pas pouvoir reconstruire le site à l'identique facilement. Néanmoins, cette méthode peut être utile pour une mise à jour rapide.

La démarche d'installation d'un plugin est alors la suivante :

- Arrêter la Webapp
- Dézipper la distribution binaire du plugin dans le répertoire de la webapp
- [facultatif] Exécuter les scripts de création et/ou d'initialisation des tables associées au plugin (Cf. ci-dessous pour localiser ces scripts dans les différentes distributions)
- [facultatif] Configurer le fichier properties du plugin
- [facultatif] Configurer le fichier db.properties de Lutece pour ajouter un nouveau pool de connexions spécifique au plugin
- Relancer la Webapp
- Se connecter en Admin et se rendre dans la fonction Gestion des plugins
- Le nouveau plugin doit apparaitre non installé dans la liste des plugins. Cliquer sur Installer.
- [facultatif] Sélectionner le pool de connexions éventuellement utilisé par le plugin.

GÉNÉRALITÉS CONCERNANT LE DÉVELOPPEMENT SUR

LUTECE

5. CONSTRUCTION ET ASSEMBLAGE : LES BUILDS
6. RÈGLES DE CODAGE
7. RÈGLE DE NOMMAGE ET VÉRIFICATION
8. LES RÈGLES DE DOCUMENTATION
9. GESTION DES FAITS TECHNIQUES
10. GESTION DES SOURCES SUR GITHUB

5. CONSTRUCTION ET ASSEMBLAGE : LES BUILDS

Un portail web lutece est une webapp java EE. Le contenu de cette webapp (JSP, classes java, templates HTML, etc.) provient directement du coeur de Lutece (obligatoire) et de plugins (facultatifs). L'assemblage d'une webapp à partir du coeur et de la liste de plugins est faite avec l'outil **Maven** (https://maven.apache.org/). Avant d'avancer dans la suite de ce chapitre, il est recommandé de connaitre le fonctionnement de Maven.

En plus d'une utilisation classique de Maven, l'assemblage de webapps Lutece s'appuie sur:

- deux POMs parents lutece-global-pom (pour "lutece-core" et "lutece-plugin") et lutece-site-pom (pour "lutece-site"). Ces POMs regroupent les configurations par défaut de projets lutece.
- un plugin Maven: lutece-maven-plugin. Ce plugin gère l'assemblage final de la webapp.

ASSEMBLAGE D'UNE WEBAPP LUTECE

L'architecture du projet lutece est de construire une webapp contenant l'intégralité des fonctionnalités des plugins choisis. Pour ce faire, lutece-core contient les fichiers obligatoires d'une webapp (par exemple WEB-INF/web.xml) et les fichiers qui servent aux fonctionnalités communes à tous les plugins (code java, header et footer des pages html, etc.). Les plugins apportent en plus leurs propres fichiers. De plus, il est parfois nécessaire pour un site donné de pouvoir modifier certains fichiers présents dans lutece-core ou dans un lutece-plugin (par exemple, modifier un template html). L'objectif principal de l'assemblage d'une webapp lutece est donc de produire les bons fichiers venant des différentes sources à inclure dans la webapp finale. Nous allons voir comment lutece-maven-plugin atteint cet objectif.

Cependant, en dehors de l'assemblage de webapps, l'utilisation de maven permet aussi d'atteindre d'autres objectifs: gérér la compilation et les tests du code, pouvoir publier des artifacts fournissant une fonctionnalité réutilisable, générer la documentation technique (javadoc et Maven Site), etc.

LUTECE-MAVEN-PLUGIN

lutece-maven-plugin effectue les actions suivantes:

- il compile le code et le place dans le dossier "WEB-INF/classes" de la webapp: dossier "src/"
- il copie les fichiers de la webapp à la racine de la webapp: dossier "webapp/"
- il copie le contenu (du code compilé ou des fichiers de webapp) des dépendances.

Ces actions sont utilisées dans plusieurs contextes: Lutece-maven-plugin définit en effet plusieurs types d'artifacts maven qui correspondent aux différents éléments de l'architecture lutece: lutece-core, lutece-plugin, lutece-site. Lutece-core et lutece-plugin sont des artifacts qui contiennent des classes java à compiler et des fichiers à copier dans la webapp, lutece-site contient uniquement des fichiers à copier dans la webapp, surchargeant ceux provenant d'artifacts lutece-core ou lutece-plugin. Lutece-maven-plugin fournit des goals maven pour obtenir une webapp en version "exploded" pour les artifacts de type lutece-core et lutece-plugin, en version "war" pour les artifacts de type lutece-site. Nous verrons ces goals maven juste après.

La webapp complète contiendra aussi les fichiers provenant de deux dossiers spéciaux: defaultConfDirectory et localConfDirectory:

- defaultConfDirectory a pour valeur par défaut "src/conf/default". Lutece-site-pom définit des profils maven qui changent la valeur de defaultConfDirectory à "src/conf/<profile>" : "dev", "rec", "integ", "formation", "preprod", "prod".
- localConfDirectory a pour valeur par défaut "$HOME/lutece/conf/<artifactId>". Attention, ces fichiers ne proviennent pas d'un artifact maven, ils provoquent donc des assemblages non reproductibles. Il faut les utiliser avec parcimonie.

Enfin, le reactor maven (mode multi modules) est géré de manière très différente d'un projet maven multi module classique, comme nous le verrons après.

RÈGLES D'UTILISATION DE LUTECE-MAVEN-PLUGIN

LES GOALS MAVEN UTILISÉS

Nous utiliserons les goals suivants de lutece-maven-plugin :

- lutece:exploded pour lutece-core et lutece-plugin
- lutece:site-assembly pour lutece-site

ORDRE DE SURCHARGE DES FICHIERS

Lutece-maven-plugin utilise de manière pervasive un concept similaire aux overlays de maven-war-plugin. Il est important de savoir dans quel ordre les overlays sont appliqués pour savoir quel fichier est utilisé lorsqu'un fichier est présent dans plusieurs sources. Ceci concerne uniquement les fichiers de la webapp (par exemple dans le dossier "webapp/" d'un lutece-plugin) et un assemblage sans modules maven. L'ordre est :

1. dépendances de type lutece-core ou lutece-plugin
2. dossier "webapp" de l'artifact courant
3. defaultConfDirectory de l'artifact courant
4. localConfDirectory de l'artifact courant

La version utilisée sera celle présente dans l'élément de cette liste avec le plus haut numéro.
Note: Il n'y a pas d'ordre défini entre les fichiers de deux dépendances, il ne faut donc pas qu'un même fichier existe dans deux dépendances différentes.

Pour l'assemblage en multi-module, l'ordre dépend de l'ordre des artifacts dans le reactor.

EXEMPLES D'UTILISATION DE LUTECE-MAVEN-PLUGIN

ASSEMBLAGE D'UN LUTECE-SITE

L'utilisation la plus simple de lutece-maven-plugin consiste à construire un artifact de type lutece-site sans plugins ni surcharge. Le projet contient alors uniquement un fichier pom.xml à la racine. Ce POM contient:

- La définition des dépôts maven de lutece (pour obtenir le POM parent et le plugin maven)
- le lien vers le POM parent
- le packaging de l'artifact: lutece-site
- la dépendance vers l'artifact lutece-core
- Maven va télécharger l'artifact lutece-core depuis les dépôts et fabriquer la webapp à partir de cet artifact.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/maven-v4_0_0.xsd">
    <parent>
        <artifactId>lutece-site-pom</artifactId>
        <groupId>fr.paris.lutece.tools</groupId>
        <version>2.0.4</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>lutece-example</artifactId>
    <packaging>lutece-site</packaging>
    <name>Lutece example</name>
    <version>1.0.0</version>
    <repositories>
        <repository>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
            <id>luteceSnapshot</id>
```

```
            <name>luteceSnapshot</name>
            <url>http://dev.lutece.paris.fr/snapshot_repository</url>
        </repository>
        <repository>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
            <id>lutece</id>
            <name>luteceRepository</name>
            <url>http://dev.lutece.paris.fr/maven_repository</url>
        </repository>
    </repositories>
    <dependencies>
        <dependency>
            <groupId>fr.paris.lutece</groupId>
            <artifactId>lutece-core</artifactId>
            <version>5.0.0</version>
            <type>lutece-core</type>
        </dependency>
    </dependencies>
</project>
```

On assemble la webapp en utilisant le goal maven :

```
$ mvn lutece:site-assembly
```

Cela produit la webapp en version "war" et "exploded" dans le dossier target:

```
lutece-example-1.0.0 lutece-example-1.0.0.war
```

On peut aussi activer des profils maven lors de ce goal pour utiliser un defaultConfDirectory différent :

```
$ mvn lutece:site-assembly -P prod
```

ASSEMBLAGE D'UN LUTECE-PLUGIN OU DE LUTECE-CORE

En utilisant le pom.xml du projet qu'on veut assembler, par exemple pour *https://github.com/lutece-platform/lutece-core*, on peut produire en invoquant le goal maven :

```
$ mvn lutece:exploded
```

La webapp en version "exploded" est disponible dans le dossier target/lutece/

UTILISATION DU DÉPÔT MAVEN LOCAL

Pour simplement propager les changements faits sur un artifact dans une webapp assemblée à partir d'un autre artifact, on peut utiliser le dépôt maven local comme intermédiaire.

Dans le premier projet, utiliser le goal:

```
$ mvn install
```

Puis dans le deuxième projet, utiliser :

```
$ mvn clean lutece:exploded
```

Cette méthode présente l'avantage de la simplicité, mais elle conduit à installer dans son dépôt local maven des artifacts qui peuvent être différents de ceux publiés dans les dépôts lutece. Cela peut rendre le travail collaboratif difficile car on ne teste plus la même chose que les autres. Il faut donc l'utiliser avec parcimonie, ou alors parfaitement maitriser le contenu de son dépôt local (par exemple en supprimant les artifacts installés à la main).

UTILISATION DU MAVEN REACTOR (MULTI MODULE)

Pour éviter d'avoir à utiliser maven install sur toutes les dépendances, de polluer son dépôt local avec des versions différentes d'artifacts, on peut utiliser les modules maven et le reactor. Attention, l'utilisation des modules maven est très particulière avec lutece-maven-plugin et ne s'utilise qu'avec le goal "lutece:exploded". Le mode multi-module assemble une webapp à partir des différents artifacts du reactor dans le dossier "target/lutece" au niveau du pom aggrégateur. On ne peut donc pas utiliser d'artifact lutece-site dans ce genre d'assemblage. Contrairement à une utilisation classique du reactor où les derniers modules produisent les artifacts désirés, lutece-maven-plugin assemble directement tous les modules dans le dossier target du pom aggrégateur au cours du traitement du reactor.

On peut utiliser la structure de projets suivante où on a les différents projets venant de github et le global-pom :

```
lutece-core lutece-search-library-lucene lutece-system-plugin-systeminfo pom.xml
```

Par convention, on définit des modules lorsque le profile "multi-project" est activé en rajoutant ceci au global-pom pom.xml :

```
<profiles>
 <profile>
 <id>multi-project</id>
 <modules>
 <module>lutece-core</module>
 <module>lutece-system-plugin-systeminfo/</module>
 <module>lutece-search-library-lucene</module>
 </modules>
 </profile>
</profiles>
```

On peut ensuite assembler la webapp avec le goal:

```
 $ mvn lutece:exploded -P multi-project
 [INFO] ------------------------------------------------------------------------
 [INFO] Reactor Summary:
 [INFO]
 [INFO] Lutece global pom ................................ SUCCESS [1.238s]
 [INFO] Lutece Analyzers and indexers .................... SUCCESS [2.000s]
 [INFO] Lutece ........................................... SUCCESS [4.659s]
 [INFO] Lutece systeminfo plugin ......................... SUCCESS [0.325s]
 [INFO] ------------------------------------------------------------------------
 [INFO] BUILD SUCCESS
 [INFO] ------------------------------------------------------------------------
```

La webapp est disponible dans le dossier "**target/lutece**"

Note: attention, à cause des particularités de lutece-maven-plugin en multi-modules, pour nettoyer et recompiler tout en une seule commande on doit utiliser :

```
$ mvn lutece:clean lutece:exploded -P multi-project
```

Enfin, pour ne recompiler qu'un seul projet d'un build multi module, on peut utiliser l'option testWebappDirectory en se plaçant au niveau du pom du projet en question:

```
$ mvn lutece:exploded -DtestWebappDirectory="<path to top multiproject>/target/lutece"
```

6. RÈGLES DE CODAGE

Pour faciliter sa lecture, le code doit respecter des règles de mise en forme appliquées à l'ensemble des sources. Les règles ci-dessous sont celles appliquées depuis l'origine du projet.

MISE EN FORME ET PRÉSENTATION DU CODE JAVA

Les principales règles concernant le code Java dans Lutece sont :

- Accolades sur une nouvelle ligne (Style C/C++)
- Indentation de 4 caractères

La présentation générale des fichiers devra respecter celle existante (pour les classes java la présentation des commentaires, indentation du code, utilisation de l'anglais.)

L'ensemble de l'application doit conserver une homogénéité complète tant sur la présentation du code que sur les règles de nommages, l'utilisation de l'anglais, l'insertion de commentaires.

Tous les fichiers sources doivent comporter la licence de diffusion en entête.

La mise en forme du code doit se faire via **Maven** à l'aide de l'outil **Jalopy** dont les règles sont référencées dans les fichiers POM parents.

Voici la commande à lancer à la racine du projet :

```
$ mvn jalopy:format
```

LE CODE HTML

Le code HTML doit être correctement indenté et respecter la norme HTML 5.0 (*http://www.w3.org/TR/html5/*).

Les fichiers doivent être encodés en UTF-8.

LE JAVASCRIPT

L'usage du javascript doit être limité au maximum en raison des problèmes d'accessibilité engendrés et des problèmes liés aux utilisateurs qui désactivent l'exécution des scripts dans leurs navigateurs.

Attention : *L'application doit impérativement rester opérationnelle lorsque les scripts sont désactivés.*

LES STYLES CSS

Tous les attributs de mise en forme du code HTML doivent être gérés par des feuilles de style CSS3 dont les spécifications sont définies par le W3C (*http://www.w3.org/Style/CSS/*).

Les styles devront utiliser au maximum l'héritage et les surcharges devront être limitées au maximum.

REQUÊTE SQL : NORME SQL-92

Toutes les requêtes SQL (présentes dans les scripts SQL ou dans les DAO) doivent suivre la liste de règles ci-dessous afin de respecter au maximum la norme SQL-92. Tous les formats de données ou syntaxes spécifiques à un Système de Gestion de Base de Données (SGBD) doivent être évités afin de garantir la possibilité d'utilisation de Lutèce avec différents SGBD.

Description	**Exemple de syntaxe SQL spécifique (MySQL)**	**Syntaxe SQL-92 équivalente à utiliser**
Les caractères anti-côtes utilisés pour les noms de tables ou de colonnes doivent être supprimés	CREATE TABLE `core_admin_auth_db_module`	CREATE TABLE core_admin_auth_db_module
La définition de l'encodage pour une colonne ou une table doit être supprimée	CREATE TABLE ... (access_code VARCHAR(16) collate utf8_unicode_ci,	CREATE TABLE ... (access_code VARCHAR(16),
Lors de la création d'une table, pour une colonne, la déclaration de la valeur par défaut et le fait que cette colonne puisse stocker une valeur NULL doit respecter un ordre donné	CREATE TABLE ... (access_code VARCHAR(16) NOT NULL DEFAULT	CREATE TABLE ... (access_code VARCHAR(16) DEFAULT '' NOT NULL,
Lors de la création d'une table, le moteur de stockage ainsi que l'encodage ne doivent pas être spécifiés	CREATE TABLE ... (...) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;	CREATE TABLE ... (...);
Les tailles des champs de type entier (INT, SMALLINT, ...) ne doivent pas être spécifiées	CREATE TABLE ... (id_mailinglist INT(11)NOT NULL DEFAULT '0',	CREATE TABLE ... (id_mailinglist INT NOT NULL DEFAULT '0',
Suppression des types de données non signées.	CREATE TABLE ... (id_mailinglist INT UNSIGNEDNOT NULL DEFAULT '0',	CREATE TABLE ... (id_mailinglist INT NOT NULL DEFAULT '0',
Déclaration des index de manière explicite et non lors de la définition de la table	CREATE TABLE core_admin_right (... , KEY index_right (level_right, admin_url))	CREATE TABLE core_admin_right (...);CREATE INDEX index_right ON core_admin_right (level_right, admin_url);
Ne pas utiliser la fonctionnalité ON UPDATE CURRENT_TIMESTAMP permettant de mettre à jour un champ date lors de la mise à jour d'un tuple.	CREATE TABLE ... (date_login TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL ON UPDATE CURRENT_TIMESTAMP);	CREATE TABLE ... (date_login TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL);
Ne pas utiliser le types de données TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT. Attention utiliser LONG VARCHAR écrit en deux mots.	TINYTEXT;MEDIUMTEXT;LONGTEXT	VARCHAR(255)LONG VARCHARLONG VARCHARLONG VARCHAR
Ne pas utiliser le types de données TINYBLOB, BLOB, LONGBLOB	TINYBLOB BLOB LONGBLOB	VARBINARYLONG VARBINARYLONG VARBINARYLONG
Ne pas utiliser le type TINYINT	CREATE TABLE ... (status TINYINT);	CREATE TABLE ... (status

Ne pas utiliser le type TINTINT	CREATE TABLE ... (status TINTINT);	SMALLINT);
Suppression des commentaires MySQL générés lors de l'export d'une table par exemple.	/*!40101 SET NAMES utf8 */;	aucun commentaire
Ne pas utiliser le caractère d'échappement antislash. Pour échapper une simple cote, il faut doubler chaque simple cote. Ne pas utiliser de double cote pour délimiter un champ, utiliser une simple cote.	L\'accès aux ressourcesINSERT INTO core_admin_right VALUES (contenu)Retour chariot\r\nNouvelle ligne	L''accès aux ressourcesINSERT INTO core_admin_right VALUES ('contenu')Retour chariotbr /Nouvelle ligne
Pour limiter les risques d'incompatibilité, toujours préférer stocker les données en binaire lorsque cela est possible	INSERT INTO core_stylesheet (...) VALUES ('xsl:stylesheet version=1.0...')	INSERT INTO core_stylesheet (...) VALUES (0x3C3F786D6C207665)

COMMENTAIRES

L’ensemble des éléments de l’application (programmes, scripts, fichiers de propriétés, ...) devra être commenté. Les commentaires devront être rédigés en anglais.

Pour les programmes Java, l’ensemble des classes et de leurs méthodes (y compris protected et private) devra comporter des commentaires Javadoc contenant une description de la fonctionnalité prise en charge par la méthode, ainsi que les tags **@param @return @exception** .

Les modifications réalisées dans les versions successives seront indiquées par des tags **@version** .

Les nouvelles méthodes et classes des API indiqueront leur version d'introduction à l'aide de la balise **@since** .

Les methodes obsolètes seront identifiées à l'aide de la balise **@deprecated** .

RESPECT DES NORMES D'ACCESSIBILITÉ

Dans le cadre de l'article 47 de la loi n°2005-102 du 12 février 2005, les collectivités territoriales doivent rendre accessibles leurs sites web à chaque citoyen quel que soit leur matériel ou logiciel, leur infrastructure réseau, leur langue maternelle, leur culture, leur localisation géographique, ou leurs aptitudes physiques ou mentales.

Article 47 de la loi n° 2005-102

Les services de communication publique en ligne des services de l'Etat, des collectivités territoriales et des établissements publics qui en dépendent doivent être accessibles aux personnes handicapées. L'accessibilité des services de communication publique en ligne concerne l'accès à tout type d'information sous forme numérique quels que soient le moyen d'accès, les contenus et le mode de consultation. Les recommandations internationales pour l'accessibilité de l'internet doivent être appliquées pour les services de communication publique en ligne. Les règles décrites à l’annexe 6.4 rappellent la liste des règles relatives à l’accessibilité établies par l’Agence pour le Développement de l’Administration Electronique (ADAE)

Lutece est un outil réalisé et utilisé par une collectivité, et à ce titre intègre dans ses règles de développements les contraintes liées au respect du Référentiel Général d'Accessibilité des Administrations (RGAA).

Sont concernés, les sites Internet et Intranet (ces derniers étant des outils de travail courants pour les salariés handicapés, lesquels doivent pouvoir accéder à toutes les informations nécessaires à l’exécution de leurs missions), et les applications web. Par application web, on entend toute application « métier » à laquelle on accède au moyen d’un navigateur web et qui vise tout ou partie des agents d’une administration ou le grand public (exemple : inscription aux concours, logiciel de gestion de congés…).

Les impacts sur la qualité du code :

Une partie des règles d'accessibilité du web concernent le respect de normes de développements en vigueur dans le domaine pour permettre l'usage de :

- Navigateur de type texte
- Agrandissement d'une page dans un navigateur (touches CTRL-+, outil ZoomText)
- Raccourcis claviers
- Utilisation d'outils de synthèse vocale (JAWS, ...)

--> le code html/css/javascript des page web notamment doit absolument prendre en compte ces contraintes. L'usage du framework Twitter Boostrap3.x permet de respecter une partie de ces normes, mais il est important de s'assurer lors du développement des fonctionnalités que les problématiques d'accessibilité soient bien prises en comptes. En doublant par exemple les fonctionnalités en javascript par des fonctionnalités gérées côté serveur en cas de désactivation du javascript par les outils d'accessibilité.

les impacts sur la partie éditoriale :

- Les règles d'accessibilité imposent une cohérence dans les contenus, de la responsabilité des webmestres
Exemple : pour un article sur le cheval vainqueur du Grand Prix d'Amérique, la photo en illustration ne pourra pas se contenter d'avoir en attribut descriptif le terme « Cheval »

--> pour respecter ces normes un contrôle manuel des contenus est obligatoire, mais le développeur fait en sorte que l'outil doit permettre d'« encadrer » le gestionnaire de contenu dans ses fonctionnalités (en imposant des saisies de champs, ...)

Il est possible de contrôler la qualité du code de ses interfaces web avec des outils de contrôle intégrés dans certains navigateurs web (par exemple Opquast Desktop).

7. RÈGLE DE NOMMAGE ET VÉRIFICATION

Toutes les règles de nommage décrites ici doivent être vérifiées à l'aide des outils **Checkstyle** et **PMD**. Ces outils doivent être configurés de façon à supporter les règles de nommage spécifiques, en utilisant les fichiers de configuration fournis par le projet Lutece. La production des rapports de conformité se fait lors de la génération du Maven Site.

PRINCIPES GÉNÉRAUX

Les noms des identificateurs doivent respecter les règles suivantes:

- Etre des noms communement utilisés pour désigner l'objet
- Ne pas introduire de contresens ou de problemes d'interpretation
- Etre en langue anglaise
- Ne jamais être tronqués ou contractés

Ne pas utiliser les constantes ou les variables pour stocker des messages d'erreur en dur (utiliser les fichiers de propriétes dédiés). En régle génerale, éviter toute valeur en dur, utiliser au maximum les fichiers de propriétés .properties de l'application ou des plugins.

Ne pas utiliser de chemins absolus (les seuls chemins absolus utilisés sont stockés dans les parametres exitants du fichier config.properties).

VARIABLES

Les types de base Java doivent avoir les préfixes suivants :

Préfixe	Type
n	int
str	String
b	boolean
l	long

Les instances d'objet doivent être préfixées par le nom de la classe ou la partie représentative du nom de la classe indiquant le plus présisement possible le type d'objet.

Classe	Exemple d'instance de la classe
HtmlTemplate	tList
StringBuffer ou StringBuilder	sbXml
Date	dateBegin
List, ArrayList	listUsers
Map, HashMap	mapParameters

Mise en forme : Notation Hongroise à l'exception de l'initiale en minuscules, pas d'underscore. Les abréviations sont à éviter

Les variables d'instance (**static** ou non) doivent être préfixées par un underscore.

CONSTANTES

Toute valeur littérale de chaîne de caractères ou numérique doit être placée dans une constante. Les déclarations de constantes doivent être faites dans les premieres lignes du code source ou dans une classe spéciale dédiée aux constantes.

Les préfixes doivent indiquer la nature de la constante.

Les préfixes courants sont :

Préfixe	Type
ACTION_	Pour une action du modèle MVC
DSKEY_	Pour les clés du Datastore
JSP_URL_	Pour les url des JSP
MARK_	Pour les noms des signets à substituer dans les templates
MESSAGE_	Pour les messages ou les clés de message utilisées par Message.jsp
PARAMETER_	Pour le nom d'un parametre de la requête HTTP
PATH_	Pour les chemins des répertoires
PROPERTY_	Pour les noms de clé d'un fichier .properties
TEMPLATE_	Pour les noms des templates HTML
URL_	Pour les URLs
VIEW_	Pour une vue du modèle MVC

Mise en forme des constantes : tout en majuscules, mots séparés par des underscores.

Pour un plugin d'une certaine taille, les constantes pourront être regroupées dans une classe **fr.paris.lutece.plugins.myplugins.web.Constants**.

OBJETS DE BASE DE DONNÉES

Tous les noms de table et de colonne doivent être en minuscules en séparant les mots par des underscores. Tous les noms de table doivent être préfixés par nom du plugin_ à l'exception des tables du noyau qui sont préfixées par core_.

LES REQUÊTES SQL

Les requêtes SQL des classes DAO (couche métier) doivent être placées dans des variables de type string et doivent respecter la forme suivante:

- mots clés en majuscules (SELECT, UPDATE, WHERE , AND ...),
- noms des colonnes en minuscules,
- alias des noms de colonnes sous forme de lettres de l'alphabet

Exemple de syntaxe:

```
String strSQL = SELECT a.id_theme, a.description_theme FROM theme a,
theme newsletter b WHERE a.id theme = b.id theme and b.id newsletter = ?
```

LES FICHIERS TEMPLATES

Extension : .html

Mise en forme : tout en minuscules, mots séparés par des underscores, pas d'utilisation de tirets, langue anglaise, pas d'abréviations.

Règles de nommage pour les fichiers servant à un traitement standard:

- **create_fonctionality.html** pour l'affichage d'une page de création,
- **modify_fonctionality.html** pour une page de modification,
- **remove_fonctionality.html** pour une page permettant une suppression,
- **manage_fonctionality.html** pour l'affichage d'une page de gestion d'une fonctionnalité,
- **view_fonctionality.html** pour une page d'affichage d'une fiche,
- **confirm_fonctionality.html** pour une page de message de confirmation,
- **fonctionality_list.html** pour une page d'affichage d'une liste.
- **fonctionality_row.html** pour l'affichage d'une ligne d'une liste.

Règle de nommage des signets servant à la substitution des valeurs des Bookmarks: langue anglaise, minuscules, mots séparés par des underscores, pas d'abréviation ni de tirets. (ex: **$ {file_name}**)

FICHIERS JSP

Mise en forme : Notation Hongroise, langue anglaise, pas d'abréviation, pas de tirets,pas d'underscore;

Règles de nommages pour les fichiers servant à un traitement d'affichage standard:

- **CreateFonctionality.jsp** pour les pages servant à un traitement d'affichage de page de création,
- **ModifyFonctionality.jsp** pour les pages servant à un traitement d'affichage de page de modification ,
- **RemoveFonctionality.jsp** pour les pages servant à un traitement d'affichage de page de suppression,
- **ManageFonctionality.jsp** pour les pages servant à un traitement d'affichage de page de gestion,
- **ViewFonctionality.jsp** pour les pages servant à un traitement d'affichage de page de la fiche de la fonctionnalité,
- **ConfirmFonctionality.jsp** pour les pages servant à un traitement d'affichage de la page de confirmation,
- **FonctionalityList.jsp** pour les pages servant un traitement d'affichage de liste.

Mêmes règles de nommage pour les Jsp de traitements sans affichage, auxquelles s'ajoute le préfixe Do
(ex: **DoCreateFonctionality.jsp**, **DoModifyFonctionality.jsp**).

FICHIERS IMAGES

Règle de nommage: langue anglaise, utilisation de minuscule et d'underscores entre les mots, pas de tirets ni d'abréviation.
Les préfixes utilisés doivent correspondre à la fonction de l'image quand elle existe:

Préfixe	Type
b_	Images servant de bouton
i_	Images servant d'icône (ex : i_delete.png)
l_	Images d'une ligne

Le format des images servant à la présentation des pages doit être au format **.png**.

FICHIERS XML

Les balises XML utilisées dans l'application devront être en minuscules, séparées par des tirets lorsqu'il s'agit de mots composés, en langue anglaise, sans abréviations.

Les noms des balises devront être le plus proche possible du modèle qu'ils doivent représenter: par exemple **«portlet-id»** pour la balise stockant la valeur de l'identifiant du portlet récupéré dans la colonne **id_portlet** de la table **portlet**.

8. LES RÈGLES DE DOCUMENTATION

La documentation de Lutèce est écrite au format XML, puis est générée à l'aide de **Maven** aux formats HTML et PDF à l'aide de la commande :

```
$ mvn site
```

Un nombre limité de balises est utilisé, afin de permettre une mise en page homogène dans les deux formats.

CRÉER UNE DOCUMENTATION : LES RÈGLES DE BASE

L'encoding utilisé est UTF-8, les chapitres sont découpés en **section** et **subsection** .

Le code HTML inclu dans ces chapitres ne doit pas utiliser la balise **
** , mal interpretée lors de la génération PDF. La balise **<p>** est donc à utiliser pour effectuer des retours à la ligne.

Les tables doivent comporter au moins un titre en première ligne (**<th>**)

Les images doivent mesurer 780px de large (centrer l'image sur fond blanc), et être créées au format .gif .

EXEMPLES DE MISE EN OEUVRE

Structure globale du fichier :

```
<?xml version=1.0 encoding=UTF-8?>
    <document>
        <properties>
            <title>Lutèce : titre du document</title>
        </properties>
        <body>

            <section name=Titre de chapitre 1>
                ...................
            </section>

        </body>
    </document>
```

Un document peut contenir plusieurs sections.

Une section peut contenir du texte formaté par un paragraphe (**<p>**), ainsi qu'une ou plusieurs sous-section(s)

Exemple :

```
<section name=Titre de chapitre 1>
        <p>
            Introduction du chapitre 1
        </p>

        <subsection name=Sous-chapitre 1>
            <p>
                    texte ...
            </p>
        </subsection>

        <subsection name=Sous-chapitre 2>
            <p>
                texte ...
            </p>
        </subsection>

    </section>
```

Le résultat est le suivant :

TITRE DE CHAPITRE 1

Introduction du chapitre 1

SOUS-CHAPITRE 1

texte ...

SOUS-CHAPITRE 2

texte ...

Il est possible d'inclure une liste dans un paragraphe :

```
<p>
      <ul>
         <li>exemple de liste 1</li>
         <li>exemple de liste 2</li>
      </ul>
   </p>

   <p>
      <ol>
         <li>exemple de liste numérotée 1</li>
         <li>exemple de liste numérotée 2</li>
   </ol>
</p>
```

Résultat :

- exemple de liste 1
- exemple de liste 2
- exemple de liste numérotée 1
- exemple de liste numérotée 2

Le texte peut être formaté avec des balise du type **<strong>** , **<em>** , **<code>**

```
<p>
      <code>texte au format code</code>
   </p>
   <p>
      <strong>texte au format strong</strong>
   </p>
   <p>
      <em>texte au format em</em>
   </p>
```

Résultat :

- **texte au format code**
- **texte au format strong**
- *texte au format em*

Des exemples de code peuvent être introduit à l'aide de la balise **<pre>** :

Attention : pour que le rendu soit conforme lors de l'utilisation de la balise <pre> , il faut coller le texte à gauche, sans tenir compte de l'indentation générale du code xml du fichier.

Le résultat est :

```
<application>
    <application-class>fr.paris.lutece.plugins.securedtest.SecuredTestApp</application-class>
    <application-security-model>1</application-security-model>
 </application>
```

Pour insérer un tableau, la syntaxe est la suivante :

```
<p>
<table>
<tr> <th>Titre 1</th> <th>Titre 2</th> </tr>
<tr> <td>Colonne 1</td> <td>Colonne 2</td> </tr>
</table>
</p>
```

Ce qui donne le tableau suivant :

Titre 1	Titre 2
Colonne 1	Colonne 2

Pour afficher une copie d'écran :

```
<p> <center> <img src=images/logo lutece.gif /> </center> </p>
```

La balise **<center>** n'est prise en compte que pour la génération HTML.

Pour que l'image ne soit pas déformée dans la version PDF, une astuce consiste à créer l'image utilisée pour la génération de ce format sur une taille de 780px de large (pour un format de sortie A4), la copie d'écran étant centrée dans cette largeur.

Pour que l'image ne soit pas déformée dans la version PDF, une astuce consiste à créer l'image utilisée pour la génération de ce format sur une taille de 780px de large (pour un format de sortie A4), la copie d'écran étant centrée dans cette largeur.

Le résultat est :

DOCUMENTATION MARKDOWN POUR GITHUB

Un plugin Maven spécifique à Lutece permet de convertir la page d'accueil de la documentation au format xDoc (fichier `index.xml` en anglais) en un fichier `README.md` au format *Markdown* utilisé par **GitHub**.

Voici la commande :

```
$ mvn fr.paris.lutece.tools:xdoc2md-maven-plugin:readme
```

9. GESTION DES FAITS TECHNIQUES

La gestion des faits techniques (bugs ou évolutions) est assurée à l'aide de l'outil **JIRA** (*http://dev.lutece.paris.fr/jira*).

La création d'un compte est nécessaire pour le reporting de bug. Des autorisations supplémentaires sont également nécessaires pour les développeurs afin de pouvoir modifier le statut des faits.

Voici les règles à respecter pour son utilisation :

Règle n°1 : Tout plugin ou module doit avoir son propre projet dans JIRA

Règle n°2 : Le fichier POM du plugin ou du module doit déclarer les propriétés jiraProjectName et jiraComponentId (avec ses propres valeurs ! pas de copier-coller)

```
<!-- pom.xml -->
<properties>
      <jiraProjectName>PROJECTKEY</jiraProjectName>
      <jiraComponentId>12...</jiraComponentId>
      ...
</properties>
```

Règle n°3 : Toutes les fiches doivent être écrites en **anglais**.

Règle n°4 : Le champ **Fix version/s** est **OBLIGATOIRE** dès la création de la fiche (sinon l'anomalie n'apparait pas dans la roadmap). On choisira typiquement la dernière version *unreleased*.

Règle n°5 : Les champs **Type** et **Priority** doivent être sélectionnés avec beaucoup d'attention et de rigueur. Il est par exemple important de voir dans le changelog qu'un bug critique ou bloquant a été corrigé dans une version donnée.

Règle n°6 : Il est fortement recommandé de saisir les champs **Affects Version/s** et **Description**. Une copie d'écran ou une pile d'erreur (Stack Trace) sont souvent les bienvenues.

Règle n°7 : **Tout commit doit comporter un identifiant JIRA** dans son message (voire plusieurs dans certaines conditions). Exceptions : jalopy:format, ou modification du pom.xml n'ayant pas d'incidence fonctionnelle.

Voici un exemple de message de commit :

```
MYPLUGIN-18 : Bug fix on the size calculation
```

Règle n°8 : Les identifiants JIRA mis dans les commits doivent correspondre **au même projet** que les sources commités (pour que les modifications apparaissent dans le changelog du projet)

Règle n°9 : Les releases des versions dans JIRA doivent être toujours **synchronisées avec le repository Maven**

Règle n°10 : **Tout ce qui n'est pas dans JIRA n'existe pas !** Toute anomalie ne figurant pas ou étant incorrectement saisie dans JIRA n'est pas supposée être prise en compte.

10. GESTION DES SOURCES SUR GITHUB

Les plugins génériques de Lutece sont hébergés sur la plate-forme de développement collaborative **GitHub** à l'adresse suivante :

```
https://github.com/lutece-platform
```

La gestion des sources est donc basée sur l'outil de contrôle de version (SCM) Git.

Pour mettre à jour du code, il est impératif de bien comprendre et s'inscrire dans les choix de gestion de branches retenus pour Lutece.

GESTION DES BRANCHES

Pour les plugins génériques et le core, deux branches permanentes sont utilisées :

- la branche ***develop*** est la branche dans laquelle sont faits tous les développements.
- la branche ***master*** correspondant au code dans sa version releasée. Aucun *commit* ne doit être fait sur cette branche. Un merge de la branche *develop* est fait dans cette branche au moment des releases. Aucun *commit* ou *push* ne doit être réalisé directement sur cette branche.

Des branches temporaires peuvent être créées pour des nouvelles fonctionnalités. Leur nom correspond au code de l'évolution **JIRA**.

COMMANDES GIT

POUR UNE PETITE CORRECTION OU ÉVOLUTION

Il n'est pas nécessaire de créer une branche. La modification peut se faire directement sur la branche *develop*

```
git checkout develop
git pull origin develop
<< réalisation et commit des modifications >>
git push origin develop
```

POUR UNE CORRECTION OU ÉVOLUTION SIGNIFICATIVE

Une branche temporaire *feature* (où feature désigne le code JIRA de l'évolution ou de la correction) doit être créée. Les développements se font sur la branche *feature*. La branche est partagée sur le repository *origin*. A l'issue des développements la branche sera fusionnée (*merge*) sur la branche *develop*.

```
git checkout develop
git pull origin develop
git checkout -b JIRA-NNN
git push origin JIRA-NNN
<<réalisation et commit des modifications >>
git pull origin JIRA-NNN
git checkout develop
git pull origin develop
git merge JIRA-NNN
git push origin develop
```

POUR UNE CONTRIBUTION EXTERNE

L'auteur de la contribution doit forker le projet et créer une branche correspond au JIRA de sa contribution. Il pourra ensuite proposer sa contribution par le mécanisme de *Pull Request* de GitHub

SOCLE TECHNIQUE ET PRINCIPAUX FRAMEWORKS

11. LE MOTEUR DE TEMPLATES : FREEMARKER
12. LE FRAMEWORK CSS RESPONSIVE : BOOTSTRAP
13. L'INJECTION DE DÉPENDANCES AVEC SPRING
14. LE FRAMEWORK MVC
15. LA VALIDATION DES FORMULAIRES
16. L'INTERNATIONALISATION (I18N)
17. L'ÉCRITURE DE LOGS
18. LES TESTS UNITAIRES

11. LE MOTEUR DE TEMPLATES : FREEMARKER

Le moteur de templates Freemarker est utilisé pour toute la production des pages HTML des AdminFeatures (Back Office) et des XPages (Front).

Ce moteur permet de remplir des variables situées dans un modèle de page HTML appelé *Template HTML*. Les données servant à la valorisation du template sont fournies au travers d'un objet Java de type Map communément appelé *model*.

Voici un exemple très simple d'utilisation de ce framework.

CODE JAVA

Dans le code Java, nous allons construire un "model" contenant un objet Person. Il s'agit donc de créer un objet de type **Map**, contenant une entrée dont la clé vaut "person" et la valeur est une instance de la classe Person.

Le model est transmit, accompagné du modèle de page (template) et de la locale (pour l'i18n), au Service de Template de Lutece *AppTemplateService* qui fournira en retour un objet HtmlTemplate contenant le HTML valorisé.

La récupération du code HTML produit se fait par la méthode *getHtml()*.

```
// Use constants for templates and markers
 private static final String TEMPLATE = «template.html»;
 private static final String MARK_PERSON = «person»;


 …
 Map<String,String> model = new  HashMap<String,String>();
 Person person = new Person();
 Person.setFirstName( «John» );
 Person.setLastName( «Doe» );
 Model.put( MARK_PERSON , person );
 HtmlTemplate t = AppTemplateService.getTemplate( TEMPLATE, locale , model );
 t.getHtml();
```

CODE HTML

Du côté HTML, nous pouvons utiliser l'objet "person" (clé de la Map du model) et accéder aux atrributs de la classe "Person" (objet associé à la clé) de la manière suivante :

```
 <b>First name : </b> ${person.firstName} <br/>
 <b>Last name : </b> ${person.lastName} <br/>
```

Il s'agit là d'un exemple très simple et très courant de Freemarker. Ce framework permet de faire énormément de manipulation de données au niveau des templates. Nous vous invitons vivement à consulter le site **http://freemarker.org/** pour vous rendre compte de l'ensemble des possibilités offertes par cet outil.

12. LE FRAMEWORK CSS RESPONSIVE : BOOTSTRAP

Depuis la version 4.0 sortie en octobre 2012 , Lutece s'appuie sur le framework CSS Bootstrap diffusé en Open Source par la société Twitter. Ce framework a la particularité de gérer le *responsive design* (conception adaptative en traduction litérale) , c'est à dire la capacité d'une page HTML à s'adapter à l'environnement dans lequel elle est consultée. La même page ne s'affichera pas de la même façon sur un téléphone mobile, une tablette ou sur un écran de grande dimension. Ces mécanismes s'appuie notamment sur la largeur de la page.

Les versions 4.x de Lutece utilisent des versions 2.x de Bootstrap. Twitter a conçu la version 3 de son framework en rupture de compatibilité avec la version 2, ce pour améliorer significativement les possibilités "*responsives*". La prise en compte de la nouvelle version 3 de Bootstrap dans Lutece v5 a nécessité la réalisation d'une feuille de style CSS de compatibilité de manière à limiter les problèmes d'incompatibilité.

En version 5 de Lutece, il a également été ajouté une feuille de style complémentaire, Admin-LTE, pour étendre les possibilités graphique dans le back office.

Version Lutece	Back Office	Front Office
v4	Bootstrap v2	Bootstrap v2 par défaut
v5	CSS de compatibilité Bootstrap v2/v3 + AdminLTE	Bootstrap v3 par défaut

NB : La feuille de style CSS du Front Office est une proposition par défaut. Dans la pratique, un Front Office Lutece peut intégrer n'importe quelle charte graphique.

Enfin, Lutece v5 prévoit des macros Freemarker pour l'écriture de blocs de code HTML courants (inputs, box, ...) de manière à créer un niveau d'abstraction et garantir la portabilité du code HTML au fil des évolutions des versions de Bootstrap. Il est fortement recommandé d'utiliser ces macros décrites ci-dessous.

UTILISATION DE BOOTSTRAP DANS LE BACK OFFICE LUTECE V5

Une page d'administration Lutece à partir de la version 5 utilise **Bootstrap 3** + **AdminLTE** et doit respecter les contraintes suivantes:

Les blocs de code HTML doivent être dans une section `row / col / box` (exemple en pleine largeur).

BLOC ADMIN

```
<div class="row" >
    <div class="col-xs-12 col-sm-12 col-md-12">
        <div class="box [box-solid] [box-primary | box-success | box-info | box-danger | box-warning]">
            <div class="box-header">
                <h3 class="box-title"># i18n{myplugin.feature.title}</h3>
                [ <div class="box-tools pull-right">...</div> ]
            </div>
            <div class="box-body">
                ...contenu...
            </div>
        </div>
    </div>
</div>
```

Ecriture de ce bloc avec les macros FreeMarker Lutece v5

```
<@rowBoxHeader i18nTitleKey="myplugin.feature.title" >
        ...
</@rowBoxHeader>
```

ou si il y a des boutons d'actions au niveau de la barre d'outils (en haut à droite) il faut utiliser les macros **@rowBox**, **@boxHeader** et **@boxBody** au lieu de **@rowBoxHeader**.

```
<@rowBox>
<@boxHeader i18nTitleKey="myplugin.feature.title">
    <@headerButtons>
        <form class="form-inline pull-right spaced" method="post" action="jsp/admin/plugins/myplugins/MyAction.jsp">
            ....
        </form>
    </@headerButtons>
</@boxHeader>
<@boxBody>
    ....
</@boxBody>
</@rowBox>
```

Les boutons, liens d'actions relatifs aux paramètres avancés ou à l'ajout de contenus doivent se situer en haut à droite de l'écran dans la balise <div class="box-tools"> du **box-header**.

LES ACTIONS

le formulaire doit hériter des classes css bootstrap **form-inline** et **pull-right**

```
<div class="box-tools pull-right">
   <form class="form-inline pull-right" method="post" action="jsp/admin/plugins/plugin/ManageAdvancedParameters.jsp">
         <button class="btn btn-primary btn-xs btn-flat" type="submit" >
              <span class="fa fa-cog" ></span>
                <span class="hidden-xs" ></span>
         </button>
    </form>
    <a class="btn btn-default btn-xs btn-flat" title="${action.description}" >
      <span class="fa fa-close"></span>
    </a>
</div>
```

LES BOUTONS

Les boutons, qu'ils soient sous forme de lien, balise <a>, ou de boutons de formulaire, balise <button>, doivent avoir les propriétés suivantes:

- Ils doivent tous hériter à minima des classes **btn** et **btn-default** pour un bouton gris standard.
- Ils doivent contenir une icône **fa** (font awesome) balise ou <i> (spécifique à Bootstrap) : exemple pour un bouton *ajout d'un contenu*.
- Le libellé si un peu long doit être masqué en mode téléphone mobile en utilisant la class "hidden-xs" placés dans une balise <span>
- Les bouton d'actions sont généralement de classe **btn-primary**. Les exceptions sont pour les boutons Retour ou Annuler classe **btn-default**, et les boutons Supprimer qui héritent eux de la classe **btn-danger**. On peut trouver aussi btn-success, btn-info ou btn-warning.
- La taille des bouton est gérée par les classes **btn-xs** que l'on trouvera dans les **box-tools**, **btn-sm**, **btn-lg** et **btn-block**.

Bouton de modification	fa fa-pencil
Bouton de suppression	fa fa-trash
Bouton d'activation	fa fa-check
Bouton de validation	fa fa-check
Bouton d'annulation	fa fa-close

LES MODÈLES DE PAGE D'ADMINISTRATION

Les sections suivantes donnent un aperçu des modèles standard d'interface back office.

Elles abordent également l'utilisation des macros freemarker V5 facilitant l'écriture de ces dernières.

Le modèle Manage

Ce modèle est le modèle standard d'accueil d'un plugin. Il affiche généralement la liste des ressources, avec les actions associées dans un tableau.

Les boutons d'ajout d'une ressource et de paramètres avancés sont placés en haut de la page.

La liste des ressources est affichée dans un tableau ayant les class css **table table-condensed table-striped**.

```
<div class="row">
    <div class="col-xs-12 col-sm-12 col-md-12">
        <div class="box box-primary">
            <div class="box-header">
                <h3 class="box-title"># i18n{calendar.manage_calendars.tableLabel}</h3>
                <div class="box-tools">
                    <form class="form-inline pull-right spaced" method="post"
action="jsp/admin/plugins/calendar/CreateCalendar.jsp">
                        <button class="btn btn-xs btn-flat btn-primary" type="submit" >
                            <i class="fa fa-plus"></i>
                            <span class="hidden-phone" ># i18n{calendar.manage_calendars.buttonAddCalendar}</span>
                        </button>
                    </form>
                    <#if permission_advanced_parameter>
                    <form class="form-inline pull-right" method="post"
action="jsp/admin/plugins/calendar/ManageAdvancedParameters.jsp">
                        <button class="btn btn-xs btn-flat btn-primary" type="submit" >
                            <i class="fa fa-cog"></i>
                            <span class="hidden-phone" >#
i18n{calendar.manage_calendars.labelAdvancedParameters}</span>
                        </button>
                    </form>
                    </#if>
                </div>
            </div>
            <@paginationAdmin paginator=paginator combo=1 form=0 />
            <div class="box-body table-responsive">
                <table class="table table-hover table-condensed">
                    <thead>
                        <tr>
                            <th># i18n{calendar.manage_calendars.columnTitleLabel}</th>
                            <th># i18n{calendar.manage_calendars.columnTitleActions}</th>
                        </tr>
                    </thead>
                    <tbody>
                    <tr>
                        <td>
                            ...
                        </td>
                        <td>
                           <a class="btn btn-primary btn-xs btn-flat pull-left spaced" href="#"  title="#
i18n{calendar.modify_calendar.pageTitle}" >
                                <span class="fa fa-pencil"></span>
                           </a>
                          ...
                        </td>
                    </tr>

                    </tbody>
                </table>
            </div>
          <@paginationAdmin paginator=paginator combo=0 form=1 />
        </div>
    </div>
</div>
```

Ecriture de ce bloc avec les macros FreeMarker Lutece v5

```
<@rowBox>
    <@boxHeader i18nTitleKey="calendar.manage_calendars.tableLabel">
        <@headerButtons>
                    <form class="form-inline pull-right spaced" method="post"
action="jsp/admin/plugins/calendar/CreateCalendar.jsp">
                        <button class="btn btn-xs btn-flat btn-primary" type="submit" >
                            <i class="fa fa-plus"></i>
                            <span class="hidden-phone" ># i18n{calendar.manage_calendars.buttonAddCalendar}</span>
                        </button>
                    </form>
                    <#if permission_advanced_parameter>
                    <form class="form-inline pull-right" method="post"
action="jsp/admin/plugins/calendar/ManageAdvancedParameters.jsp">
                        <button class="btn btn-xs btn-flat btn-primary" type="submit" >
                            <i class="fa fa-cog"></i>
                            <span class="hidden-phone" >#
i18n{calendar.manage_calendars.labelAdvancedParameters}</span>
                        </button>
                    </form>
                    </#if>
        </@headerButtons>
    </@boxHeader>
    <@boxBody>
          <@paginationAdmin paginator=paginator combo=1 form=0 />
          <@table>
                     <tr>
                            <th># i18n{calendar.manage_calendars.columnTitleLabel}</th>
                            <th># i18n{calendar.manage_calendars.columnTitleActions}</th>
                     </tr>
           <@tableHeadBodySeparator />
                    <tr>
                        <td>
                            ...
                        </td>
                        <td>
                           <a class="btn btn-primary btn-xs pull-left spaced" href="#"  title="#
i18n{calendar.modify_calendar.pageTitle}" >
                                <span class="fa fa-pencil"></span>
```

```
                            </a>
                           ...
                        </td>
                    </tr>
          </@table>
          <@paginationAdmin paginator=paginator combo=0 form=1 />
     </@boxBody>
</@rowBox>
```

Le modèle Create et le modèle Modify

Ce modèle est déstiné à être utilisé dans les interfaces de création et de modification des ressources

```
<div class="row">
    <div class="col-xs-12 col-sm-12 col-md-12">
        <div class="box box-primary">
            <div class="box-header">
                <h3 class="box-title"># i18n{ria.create_activity.pageTitle}</h3>
            </div>
            <div class="box-body">
                <form class="form-horizontal" method="post" name="create_calendar"
action="jsp/admin/plugins/calendar/DoCreateCalendar.jsp">
                <div class="form-group">
                    <label class="control-label col-xs-12 col-sm-3 col-md-3" for="calendar_name"> #
i18n{calendar.create_calendar.name} *</label>
                    <div class="col-xs-12 col-sm-8 col-md-6">
                        <input type="text" class="form-control input-sm" id="calendar_name" name="calendar_name"
tabindex="1" maxlength="250" >
                    </div>
                </div>
                ...
                <div class="form-group">
                    <div class="col-xs-12 col-sm-offset-3 col-sm-8 col-md-offset-3 col-md-8">
                        <button class="btn btn-primary btn-flat" type="submit" >
                            <span class="fa fa-check"></span> # i18n{portal.util.labelValidate}
                        </button>
                        <a class="btn btn-default btn-flat" href="jsp/admin/plugins/calendar/ManageCalendars.jsp" >
                            <span class="fa fa-close"></span> # i18n{portal.util.labelCancel}
                        </a>
                    </div>
                </div>
                </form>
            </div>
        </div>
    </div>
</div>
```

Ecriture de ce bloc avec les macros FreeMarker Lutece v5

```
<@rowBoxHeader i18nTitleKey="ria.create_activity.pageTitle" >
    <form class="form-horizontal" method="post" name="create_calendar"
action="jsp/admin/plugins/calendar/DoCreateCalendar.jsp">
        <@fieldInputText i18nLabelKey="calendar.create_calendar.name" inputName="calendar_name" mandatory=true
maxlength=250 />
        ...
        <@actionButtons url2='jsp/admin/plugins/calendar/ManageCalendars.jsp' />
    </form>
</@rowBoxHeader>
```

LES DASHBOARDS

Au niveau de l'accueil du back-office, les dashboard utilisent des **box** comme vu plus haut. Il peut y avoir des styles particuliers avec 'small-box" et un fond de couleur bg-info par exemple ajouter pour certains dashboards.

```
<div class="small-box bg-red">
    <div class="inner">
        <h3>${memory}</h3>
        <p> Mémoire en Mo (Utilisée / Max.)</p>
    </div>
    <div class="icon">
        <i class="fa fa-tasks"></i>
    </div>
    <a href="${url}" class="small-box-footer">
          Mémoire en Mo (Utilisée / Max.)<i class="fa fa-arrow-circle-right"></i>
    </a>
</div>
<div class="small-box bg-red">
    <div class="inner">
        <#list connections as connection>
            <h3>${connection.name}</h3>
            <p>${connection.code} Connexions (Utilisées / Max.)</p>
        </#list>
    </div>
    <div class="icon">
        <i class="fa fa-database"></i>
    </div>
    <a href="${url}" class="small-box-footer">
          Connexions (Utilisées / Max.)<i class="fa fa-arrow-circle-right"></i>
    </a>
</div>
```

4

Nombre d'utilisateurs

Gestion des utilisateurs

47

Nombre de plugins installés

Gestion des Plugins

Termes les plus recherchés

Termes les plus recherchés

Administration du Site

Dernière page modifiée

1 Home

Administration du Site

Administration du Site

Dernière rubrique modifiée

Contents available - Html ou Texte libre

Administration du Site

Mon dernier document

22 - L'équipe Lutece

Article - **Etat :** En attente de validation -

Dernière action le : 12 août 2015 12:42:32

Gestion de documents

Liste des annuaires

Titre	Actions

Liste des annuaires

Liste de formulaires

Titre	Actions
Qu'attendez vous d'un CMS open-source ? Nombre de réponses 0	

Liste de formulaires

Liste des workflows

Titre	Actions

Liste des workflows

13. L'INJECTION DE DÉPENDANCES AVEC SPRING

Le framework Spring http://spring.io est utilisé très largement pour l'injection de dépendances dans Lutece.

Cette pratique de développement, aussi appelée inversion de contrôle (IoC), a pour but de créer dynamiquement (injecter) les dépendances entre les différentes classes en s'appuyant sur une description (le fichier de *context* pour Spring). Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution. Ce découplage permet de substituer une implémentation donnée par une autre par simple modification du fichier de configuration.

Ce choix technique confère à Lutece des possibilités d'adaptabilité très grandes. Tous les services injectés par Spring peuvent être substités par d'autres implémentations sans modification du code de Lutece.

De plus, en terme de modularité, chaque plugin dispose d'un fichier de *context* propre pour déclarer ses beans. Ces fichiers sont situés dans le répertoire *WEB-INF/conf/plugins*. Ils contiennent la déclaration des beans fournis par le plugin et à charger dans le contexte global.

Le core dispose de son propre fichier de context : *WEB-INF/conf/core_context.xml*.

NB : Les noms des beans doivent être préfixés par le nom du plugin suivi d'un point pour éviter les conflits de nommage.

A titre d'exemple, tous les DAO (Data Access Object) sont par défaut « injectés » via *Spring* comme le montre le fichier *context* suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="appstore.categoryDAO"  class="fr.paris.lutece.plugins.appstore.business.CategoryDAO" />
    <bean id="appstore.applicationDAO"  class="fr.paris.lutece.plugins.appstore.business.ApplicationDAO" />
    <bean id="appstore.componentDAO"  class="fr.paris.lutece.plugins.appstore.business.ComponentDAO" />
    <bean id="appstore.iconDAO"  class="fr.paris.lutece.plugins.appstore.business.IconDAO" />
</beans>
```

L'accès à un bean se fait de la manière suivante en Java :

```
private static final String BEAN_NAME = "myplugin.mybean";
MyInterface bean = SpringContextService.getBean( BEAN_NAME );
```

NB : Il est possible d'utiliser l'annotation *@Inject* si l'on se trouve dans une classe instanciée par *Spring*.

Une pratique souvent utilisée dans Lutece, est de charger une liste de beans qui implémentent la même interface.

```
List<MyInterface> beans = SpringContextService.getBeansOfType( MyInterface.class );
```

C'est par ce mécanisme par exemple que le moteur de Workflow de Lutece est capable de charger dynamiquement toutes les tâches fournies par tous les différents plugins installés ; Celles-ci implémentant toute l'interface *Task*.

Pour plus d'informations sur le framework Spring, reportez-vous à la documentation située à l'adresse suivante : http://spring.io/docs

14. LE FRAMEWORK MVC

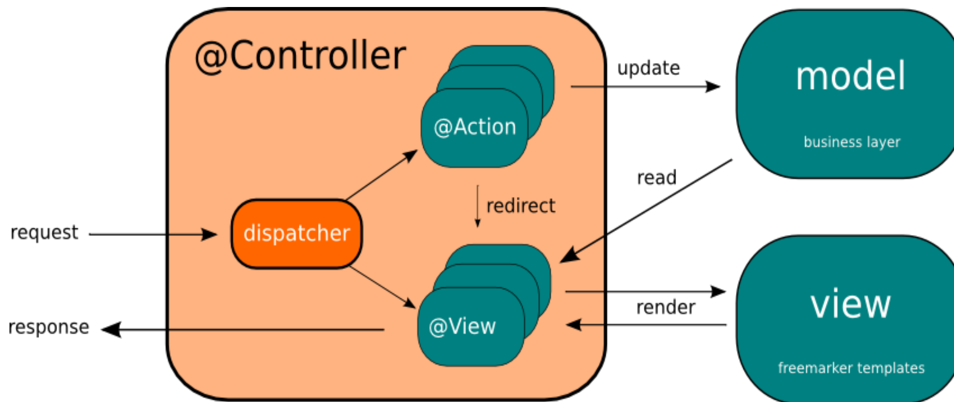
PRÉSENTATION GÉNÉRALE

Un framework utilisant le paradigme **MVC** (Model-View-Controller) a été introduit dans Lutece à partir de la version 4.1 pour développer des *XPages* (Front Office) ou des *AdminFeatures* (Back Office).

Ce framework propose :

- Le **dispatching des requêtes par le contrôleur** pour traitement des actions et affichage des vues
- L'identification des actions et des vues par **annotations**
- La population et la **validation des beans** (JSR 303)
- La gestion des erreurs de validation directement incluse dans le modèle du template (marker « errors »)
- La gestion des notifications ou messages informatifs incluse dans le modèle du template (marker « infos »)
- La redirection HTTP des requêtes vers des vues évitant le recours à des JSP

Voici la cinématique mise en oeuvre par le framework :



La mise en oeuvre de ce framework est détaillée plus loin dans les chapitres du livre traitant de la création d'une *XPage* ou d'un *AdminFeature*.

15. LA VALIDATION DES FORMULAIRES

Pour la validation des formulaires, il est fortement préconisé d'utiliser la JSR 303 (*Bean Validation*) définissant un modèle de meta-données et une API pour valider les Beans Java. Cette validation s'effectue en utilisant les annotations mais il est possible d'utiliser des fichiers XML. Cette JSR a été finalisée en novembre 2009 et fait partie de la spécification de JEE6 sortie un mois plus tard. Le support natif partiel a été introduit dans Lutece dans la version 3.0.8 avec la classe `fr.paris.lutece.util.beanvalidation.BeanValidationUtil`. Un support plus complet est disponible dans la version 4.1 que nous allons décrire ci dessous.

RAPPEL DE LE JSR 303

Voici les annotations des contraintes standards définies par la JSR :

Annotation	Description de la contrainte	Type sur lequel la contrainte peut s'appliquer
@Null	L'élément doit être nul	Object
@NotNull	L'élément doit être non nul	Object
@AssertTrue	L'élément doit être true	boolean, Boolean
@AssertFalse	L'élément doit être false	boolean, Boolean
@Min	L'élément doit être supérieur à la valeur spécifiée dans l'annotation	BigDecimal, BigInteger, byte, short, int, long
@Max	L'élément doit être inférieur à la valeur spécifiée dans l'annotation	BigDecimal, BigInteger, byte, short, int, long
@DecimalMin	L'élément doit être supérieur à la valeur spécifiée dans l'annotation	BigDecimal, BigInteger, String, byte, short, int, long
@DecimalMax	L'élément doit être inférieur à la valeur spécifiée dans l'annotation	BigDecimal, BigInteger, String, byte, short, int, long
@Size	L'élément doit être entre deux tailles spécifiées	String, Collection, Map, Array
@Digits	L'élément doit être un nombre compris dans une certaine fenêtre	BigDecimal, BigInteger, String, byte, short, int, long
@Past	L'élément doit être une date dans le passé	Date, Calendar
@Future	L'élément doit être une date dans le future	Date, Calendar
@Pattern	L'élément doit respecter une expression régulière	String

D'autres frameworks tels qu'Hibernate fournissent des contraintes intéressantes : `@CreditCardNumber`, `@Email`, `@NotBlank`, `@NotEmpty`, `@Range`, `@ScriptAssert`, `@URL`. Ces contraintes peuvent s'appliquer à un attribut, une méthode ou à la classe. Dans la majorité des cas elles seront placées au niveau de l'attribut de la manière suivante :

```
MyBean.java
@NotEmpty()
    @Pattern(regexp = "[a-z-A-Z]")
    @Size(max = 5)
    private String _strName;
    @Size(min = 10)
    private String _strDescription;
    @Min(value = 5)
    private int _nAge;
    @Email()
    private String _strEmail;
    @Past()
    private Date _dateBirth;
    @Future()
    private Date _dateEndOfWorld;
    @DecimalMin(value = "1500.0")
    private BigDecimal _salary;
    @DecimalMax(value = "100.0")
    private BigDecimal _percent;
    @Digits(integer = 15, fraction = 2)
    private String _strCurrency;
```

La vaildation du bean à l'aide d'une instance de la classe Validator produira un ensemble de violations de contraintes

```
// Check constraints
    Set<ConstraintViolation<Person>> errors = validator.validate( myBean );
    if ( errors.size(  ) > 0 )
    {
        // Handle errors
        ...
    }
```

L'IMPLÉMENTATION PROPOSÉE PAR LUTECE À PARTIR DE LA VERSION 4.1

APPORT DE L'IMPLÉMENTATION PAR RAPPORT À L'IMPLÉMENTATION DE BASE

Le principal apport de l'implémentation proposée par Lutece réside dans la gestion des messages d'erreurs. En effet celle-ci propose :

- la gestion de la locale et le support du modèle de plugin (les messages d'erreurs peuvent être définis dans les fichiers de ressources internationalisés des plugins)
- la gestion de l'affichage de la liste des erreurs pour le back office
- un modèle normalisé prévoyant l'utilisation de messages standards ou spécifiques par champs.

MESSAGES DANS LES FICHIERS RESSOURCES DES PLUGINS

L'implémentation de base de la JSR 303 propose des messages par défaut ou une personnalisation à l'aide d'un unique fichier bundle validation.properties. Ceci ne peut convenir dans le contexte de plugins de Lutece. Cependant la norme prévoit de pouvoir modifier la gestion des messages en utilisant une implémentation spécifique de l'interface MessageInterpolator. C'est le choix retenu pour Lutece. Une classe `LuteceMessageInterpolateur` a été créée afin que les messages puissent utiliser les fonctionnalités de I18nService et notamment l'écriture pour rechercher la valeur d'une clé. Les clés utilisées pour les messages peuvent ainsi être gérées à l'identique de celles des templates HTML, avec comme convention : nom_du_plugin.validation.nom_objet_metier.nom_attribut.nom_contrainte.

La mise en oeuvre de ce MessageInterpolator spécifique se fait par le biais du fichier de configuration META-INF/validation.xml.

Voici comment doivent s'écrire les contraintes de validation sur les champs d'un objet métier :

```
// Person.java

  @NotEmpty(message = "")
  @Size(max = 20, message = "")
private String _strPersonName;
```

Les clés sont déclarées dans les fichiers de ressources comme suit :

```
# myplugin_messages_fr.properties
validation.person.personName.notEmpty=Le champ 'Nom' est obligatoire. Veuillez le remplir SVP.
validation.person.personName.size=Le champ 'Nom' ne doit pas contenir plus de 20 caractères.
```

MESSAGES STANDARDS PROPOSÉS PAR L'IMPLÉMENTATION

L'implémentation propose des messages par défaut pour un certain nombre d'annotations standards ou spécifiques (Hibernate).

Ces messages sont stockés dans le fichiers ressources du core : validation_messages*.properties. En voici la liste :

Annotation	Clé	Message	Origine de l'annotation
@Size(min, max)	portal.validation.message.size	La valeur du champ {0} est invalide. Le champ doit contenir entre {1} et {2} caractères.	Standard javax.validation
@Size(min)	portal.validation.message.sizeMin	La valeur du champ {0} doit avoir une taille supérieure à {2} caractères.	Standard javax.validation
@Size(max)	portal.validation.message.sizeMax	La valeur du champ {0} doit avoir une taille inférieure à {2} caractères.	Standard javax.validation
@Pattern(regexp)	portal.validation.message.pattern	Le format du champ {0} ne respecte pas la forme {1}.	Standard javax.validation
@Min(value)	portal.validation.message.min	La valeur du champ {0} doit être supérieure à {1}.	Standard javax.validation
@Max(value)	portal.validation.message.max	La valeur du champ {0} doit être inférieure à {1}.	Standard javax.validation
@DecimalMin(value)	portal.validation.message.decimalMin	La valeur du champ {0} doit être supérieure à {1}.	Standard javax.validation
@DecimalMax(value)	portal.validation.message.decimalMax	La valeur du champ {0} doit être inférieure à {1}.	Standard javax.validation
@Digits(integer,fraction)	portal.validation.message.digits	La valeur du champ {0} doit avoir une partie entière inférieure à {1} chiffres et une partie décimale inférieure à {2} chiffres.	Standard javax.validation
@Past	portal.validation.message.past	La date du champ {0} doit être antérieure à la date du jour.	Standard javax.validation
@Future	portal.validation.message.future	La date du champ {0} doit être postérieure à la date du jour.	Standard javax.validation
@NotEmpty	portal.validation.message.notEmpty	Le champ {0} est obligatoire. Veuillez le remplir SVP.	Hibernate validators
@Email	portal.validation.message.email	La valeur du champ {0} ne correspond pas à un email valide.	Hibernate validators

L'écriture de ces contraintes dans le bean est donc de la forme suivante :

```
// MyBean.java

@NotEmpty(message = "Le champ {0} est obligatoire. Veuillez le remplir SVP.")
    @Pattern(regexp = "[a-z-A-Z]", message = "Le format du champ {0} ne respecte pas la forme {1}. ")
    @Size(max = 5, message = "La valeur du champ {0} doit avoir une taille inférieure à {2} caractères.")
    private String _strName;
    @Size(min = 10, max = 50, message = "La valeur du champ {0} est invalide. Le champ doit contenir entre {1} et {2}
caractères.")
    private String _strDescription;
    @Min(value = 5, message = "La valeur du champ {0} doit être supérieure à {1}.")
    private int _nAge;
    @Email(message = "La valeur du champ {0} ne correspond pas à un email valide.")
    private String _strEmail;
    @Past(message = "La date du champ {0} doit être antérieure à la date du jour.")
    private Date _dateBirth;
    @Future(message = "La date du champ {0} doit être postérieure à la date du jour.")
    private Date _dateEndOfWorld;
    @DecimalMin(value = "1500.0", message = "La valeur du champ {0} doit être supérieure à {1}.")
    private BigDecimal _salary;
    @DecimalMax(value = "100.0", message = "La valeur du champ {0} doit être inférieure à {1}.")
    private BigDecimal _percent;
    @Digits(integer = 15, fraction = 2, message = "La valeur du champ {0} doit avoir une partie entière inférieure à
{1} chiffres et une partie décimale inférieure à {2} chiffres.")
    private String _strCurrency;
    @URL( message = "La valeur du champ {0} ne correspond pas à une URL valide." )
    private String  strUrl;
```

AFFICHAGE DES MESSAGES

Une nouvelle classe ValidationError a été créée pour mettre en forme les messages de violation de contraintes en récupérant notamment le nom du champ et les paramètres de la contrainte.

La liste des méthodes de création de message dans le BackOffice de la classe AdminMessageService a été étendue pour recevoir en paramètres des listes d'erreurs sous la forme :

- `Set<ConstraintViolation>` pour une récupération brute des messages de contraintes ou
- `List<ValidationError>` pour bénéficier des messages standards avec les noms de champs (à privilégier)

Les noms des champs sont récupérés au travers des fichiers ressources. Les clés seront définies dans les fichiers ressources en utlisant la convention suivante model.entity.<entity>.attribute.<attribute>.

Le prefix des clés pour un bean donné est donc : `model.entity.mybean.attribute`.

Par exemple, concernant l'attribut _strPersonName de la classe Person, voici la clé du fichier ressource :

```
#myplugin_messages_fr.properties

model.entity.person.attribute.personName=Personne
```

Dans un JspBean du *back office*, voici la déclaration du préfixe et la forme que prend la validation d'un bean :

```
// PersonJspBean.java
private static final String VALIDATION_ATTRIBUTES_PREFIX = "model.entity.person.attribute.";
        ...

    public String doCreatePerson( ... )
    {
        ...
        // Check constraints
        List<ValidationError> errors = validate( person , VALIDATION_ATTRIBUTES_PREFIX );
        if (errors.size() > 0)
        {
            return AdminMessageService.getMessageUrl( request, Messages.MESSAGE_INVALID_ENTRY, errors );
        }
        ...
    }
```

OPTIONS AVANCÉES DE CONFIGURATION DES VALIDATIONERROR

Les options par défaut utilisées pour convertir et mettre en forme les `ConstraintViolation` en `ValidationError` sont définies dans une classe `DefaultValidatorErrorConfig` implémentant la classe `ValidatorErrorConfig`.

Il est possible d'étendre cette classe pour traiter de nouveaux paramètres de contraintes ou modifier le rendu du nom champ (entouré de la balise <strong> par défaut). Il faudra alors passer cette implémentation à la méthode validate à la place du préfixe (ce dernier faisant partie également des paramètres de configuration).

16. L'INTERNATIONALISATION (I18N)

PRINCIPES DE LOCALISATION DES RESSOURCES

Le mécanisme de localisation des libellés et des messages s'appuie sur les recommandations et les outils proposés par l'API Java. Les fichiers contenant les ressources textes localisées sont des fichiers "properties" ayant un suffixe précisant la langue et éventuellement le pays. La lecture des fichiers est réalisée par la classe `ResourceBundle` qui choisira en fonction de la *locale* (variable représentant une localisation basée sur une langue et un pays) parmi les fichiers suivants par exemple :

- `myplugin_messages.properties` - langage par défaut
- `myplugin_messages_fr.properties` - français
- `myplugin_messages_fr_FR.properties` - français (France)
- `myplugin_messages_fr_CA.properties` - français (Canada)
- `myplugin_messages_en_US.properties` - anglais (Etats-Unis)

RÈGLES CONCERNANT LES FICHIERS

Afin que Lutece puisse déterminer dans quel fichier rechercher une clé, le nom et l'emplacement des fichiers de ressources sont soumis aux règles suivantes selon le type de composant (élément du noyau, plugin ou module d'un plugin):

SOUS-SYSTÈME DU NOYAU

Nom du fichiers	*<sous-systeme>*_messages.properties
Emplacement du fichier	fr.paris.lutece.portal.resources
Construction de la clé	portal.*<sous-systeme>*.*<key>*

PLUGIN

Nom du fichiers	*<plugin>*_messages.properties
Emplacement du fichier	fr.paris.lutece.portal.*<plugin>*.resources
Construction de la clé	*<plugin>*.*<key>*

MODULE D'UN PLUGIN

Nom du fichiers	*<module>*_messages.properties
Emplacement du fichier	fr.paris.lutece.portal.*<plugin>*.modules.*<module>*.resources
Construction de la clé	module.*<plugin>*.*<module>*.*<key>*

LOCALISATION DES TEMPLATES

Les templates sont automatiquement localisés au moment de leur chargement par la methode `AppTemplateService.getTemplate( ... , Locale locale , ... )`. Les clés à localiser sont indiquées par le préfix `#i18n` suivi de la clé entourée d'accolades.

La structure des clés est la suivante :

Prefix de localisation du bundle	Nom du template	Nom de libellé (label, titre, bouton)
portal.*<sous-systeme>*. *<plugin>* module.*<plugin>*.*<module>*.	manage_myobjects create_myobject	titleObjectList labelName buttonCreate

```
<h2>#i18n{document.create_document.title} « ${document_type} »</h2>
 <div class="highlight-box">
<p> <label for="document_title">
#i18n{document.create_document.labelDocumentTitle}* : </label>
<input type="text" name="document_title" size="80" maxlength="255"/>
#i18n{document.create document.helpDocumentTitle} </p>
```

LOCALISATION DANS LE CODE JAVA

Dans le code Java, il faut recourir au service `I18nService` pour localiser une clé en fonction de la locale courante. NB : Les valeurs des clés doivent être déclarées dans des constantes comme dans l'exemple ci-dessous.

```
private static final
PROPERTY_MESSAGE_MYMESSAGE = "myplugin.myMessage";
... String strLocalizedMessage = I18nService.getLocalizedString (
PROPERTY MESSAGE MYMESSAGE , locale ); ...
```

17. L'ÉCRITURE DE LOGS

LES LOGS STANDARDS DE LUTECE

Lutece dispose de plusieurs fichiers de logs :

- **application.log** contient les principaux événements de l'application : chargement des services, des plugins, etc
- **error.log** contient tous les faits anormaux.

GESTION DES EXCEPTIONS

Toute exception doit être consignée dans un fichier log.

Les blocs `catch` ne doivent jamais être vides ou faire appel à `e.printStackTrace()`

Exceptions non critiques

Les exceptions ne nécessitant pas l'arrêt du traitement en cours doivent être loguées de la manière suivante

```
catch( Exception e )
{
     AppLogService.error( e.getMessage() , e );
}
```

Exceptions critiques

Les exceptions nécessitant l'arrêt du traitement (exemples : erreur SQL, problème d'accès fichier, ...) doivent lever une exception dérivée de `LuteceException` (**`PhysicalException`** , `ClientException` ...). La classe `LuteceException` réalise elle-même l'écriture dans les logs. Le traitement du bloc catch sera alors réalisé de la manière suivante :

```
catch( Exception e )
{
     throw new AppException( e.getMessage() , e );
}
```

Exceptions applicatives

Les exceptions applicatives, telles que `UserActionException` , sont les seules qui ne nécessitent pas d'être loguées car elles correspondent à un cas normal d'exécution de l'application.

LOGGERS SPÉCIFIQUES

Certaines applications ou certains services peuvent avoir recours à un logger spécifique. Cette pratique n'est pas recommandée dans la majorité des cas car elle requiert de modifier de manière spécifique la configuration des appenders dans le fichier `WEB-INF/conf/config.properties`.

18. LES TESTS UNITAIRES

Les tests unitaires automatisés offrent la possibilité de réaliser très rapidement un ensemble de contrôles sur le code. Ils permettent notamment de vérifier un certain nombre d'exigences, de règles de gestion et contrôler leur stabilité dans le temps (non régression). Le framework retenu pour la réalisation de ces tests est JUnit (http://www.junit.org)

MISE EN OEUVRE DE JUNIT

Il existe plusieurs façons de mettre en oeuvre JUnit. Il est également possible d'utiliser d'autres outils connexes. Voici celle retenue pour Lutece.

LuteceTestCase

La granularité d'un test unitaire est la classe. Pour chaque classe à tester, une classe dérivée de `LuteceTestCase` doit être créée et regroupera l'ensemble des tests à réaliser. L'arborescence des classes de test doit reproduire celle des sources de l'application.

Pour réaliser les tests de la classe :

- `src/java/`**`fr/paris/lutece/plugins/myplugin/web/MyClass`**

il faudra créer la classe :

- `test/java/`**`fr/paris/lutece/plugins/myplugin/web/MyClassTest`**

La classe `LuteceTestCase` étend la classe abstraite `TestCase` du framework JUnit en réalisant l'initialisation de l'ensemble des services de Lutèce dans l'implémentation de sa méthode `setUp()`. Il est ainsi possible, pour toutes les méthodes de test de la classe, d'accéder à des services de Lutèce : `AppTemplateService`, `AppPropertiesService`, `AppConnectionService`, etc.

Chaque méthode de test à l'intérieur de la classe est préfixée, comme l'exige le framework, par "test". Ceci permet, par introspection de retrouver dynamiquement l'ensemble des tests à lancer dans la classe.

L'objet MockHttpServletRequest

Pour réaliser certains tests, il est nécessaire de disposer d'objets de substitution (*Mock objects*) pour simuler des objets réels. Par exemple, dans la réalisation des tests de méthodes recevant un objet `request` en paramètre, il n'est pas possible de simuler un client HTTP pour fournir cet objet `request`. La classe `MockHttpServletRequest`, implémente l'interface `HttpServletRequest` et dispose d'accesseurs permettant de lui ajouter paramètres, attributs, sessions et cookies de manière à tester une situation réelle.

```
import fr.paris.lutece.test.LuteceTestCase;
import fr.paris.lutece.test.MokeHttpServletRequest;

import javax.servlet.http.Cookie;

public class MyClassTest extends LuteceTestCase
{
    /**
 * Test getNameFromCookie
 */
    public void testGetNameFromCookie(  )
    {
        MokeHttpServletRequest request = new MokeHttpServletRequest(  );
        Cookie cookieName = new Cookie( "mycookie", "name" );
        Cookie[] cookies = { cookieName };
        request.setMokeCookies( cookies );
        assertEquals( getNameFromCookie( request ) , "name" );
    }
    ...
}
```

Enchainement des tests (TestSuite)

JUnit propose un dispositif d'enchainement des tests basé sur les TestSuites. Pour Lutece, une classe `AllTests` est réalisée au niveau de chaque niveau de package et dispose d'une méthode static `suite()` renvoyant une TestSuite contenant tous les tests du package et les suites des sous packages. Ce dispositif permet de lancer les tests à n'importe quel niveau de package, en traitant automatiquement tous les tests des sous packages.

Voici un exemple d'implémentation d'une classe `AllTests`.

```
public final class AllTests
{
    /**
 * Constructor (private)
 */
    private AllTests(  )
    {
    }

    /**
 * A set of tests
 * @return Test the tests
 */
    public static Test suite(  )
    {
        TestSuite suite = new TestSuite( "Test of the package fr.paris.lutece.plugins.myplugin" );

        //$JUnit-BEGIN$
        // Add tests of the current package
        suite.addTest( new TestSuite( MyClassTest.class ) );

        // Add tests of sublevel packages
        suite.addTest( fr.paris.lutece.plugins.myplugin.business.AllTests.suite(  ) );
        suite.addTest( fr.paris.lutece.plugins.myplugin.web.AllTests.suite(  ) );

        //$JUnit-END$
        return suite;
    }
}
```

INSTALLATION ET CONFIGURATION

19. INSTALLER SON ENVIRONNEMENT DE TRAVAIL
20. TEST DE CONSTRUCTION ET DE DÉPLOIEMENT D'UN PREMIER PLUGIN
21. CONFIGURATION LUTECE AVANCÉE
22. INITIALISATION DE LA BASE DE DONNÉES

19. INSTALLER SON ENVIRONNEMENT DE TRAVAIL

Ce chapitre traite de l'installation et de la configuration de l'environnement nécessaire au developpement Lutece. Il n'a pas vocation à décrire la procédure d'installation de chaque logiciel pouvant être utilisé. Il décrit une liste des types de logiciels nécéssaires (accompagnés d'exemples les plus communément utilisés).

LISTE DES OUTILS NÉCESSAIRES

- Une base de données relationnelle. Par exemple, MySQL, MariaDB, PostgreSQL, Oracle, etc.
- Un environnement de développement web Java comportant les composants suivants:
  - JDK 7
  - Maven 3
  - Ant

  Ils peuvent être accompagnés ou fournis par un IDE tel que Eclipse, NetBeans, Intellij IDEA.

- Un conteneur de servlets. Par exemple Tomcat, Jetty, etc.
- Optionnel: Un système de gestion de versions. Le projet Lutece utilise majoritairement **git**, ainsi que **subversion**.

20. TEST DE CONSTRUCTION ET DE DÉPLOIEMENT D'UN PREMIER PLUGIN

Voici une procédure permettant de vérifier que l'environnement est correctement installé et que l'on peut récupérer le source d'un plugin sur GitHub, le compiler et le déployer.

ASSEMBLAGE DU PLUGIN-HELLOWORLD

Le **plugin-helloworld** va servir à vérifier l'environnement de développement. Dans cette partie, nous allons récupérer les sources de ce projet, l'assembler avec Maven, configurer l'accès à la base de données, initialiser la base de données, déployer la webapp dans un conteneur de servlet, activer le plugin et vérifier l'affichage du message "Hello World!"

SOURCES DU PROJET

Les sources du projet sont disponibles sur github à l'addresse *https://github.com/lutece-platform/lutece-dev-plugin-helloworld* . Il faut récupérer les sources et utiliser la version 1.0.0. On peut utiliser git ou bien un des liens de téléchargement disponibles sur *https://github.com/lutece-platform/lutece-dev-plugin-helloworld/releases/tag/1.0.0* .

ASSEMBLAGE AVEC MAVEN

Pour assembler la webapp, il faut utiliser son IDE ou maven directement pour invoquer le goal "lutece:exploded". On obtient alors la webapp dans le dossier "target/lutece".

CONFIGURATION DE LA BASE DE DONNÉES

La configuration de la base de données se trouve dans le fichier "target/lutece/WEB-INF/conf/db.properties". Il faut éditer ce fichier et fournir le nom d'utilisateur et son mot de passe sur les lignes "portal.user" et "portal.password". De plus, la ligne "portal.url" permet de choisir le nom de la base (par défaut "lutece").

INITIALISATION DE LA BASE DE DONNÉES

Pour initialiser la base de données, il faut utiliser ant avec le fichier de construction "target/lutece/WEB-INF/sql/build.xml", par exemple depuis l'IDE ou depuis une ligne de commande.

DEPLOIEMENT DE L'APPLICATION

Pour déployer l'application, il faut utiliser une des methodes de déploiement du conteneur de servlets pour deployer le dossier target/lutece, par exemple depuis l'IDE ou depuis une ligne de commande et démarrer la webapp.

ACTIVATION DU PLUGIN

Pour activer le plugin, il faut accéder à l'URL *<webapp>/jsp/admin/system/ManagePlugins.jsp* et s'authentifier en utilisant les codes d'access par défaut de l'administrateur : admin/adminadmin.

La page affiche alors la liste des plugins de la webapp, et il faut activer plugin-helloworld.

Note: selon le déploiement effectué, l'adresse de base de la webapp notée "<webapp>" ci-dessus change. Il s'agit la plupart du temps de *http://localhost:8080/lutece*.

AFFICHAGE DE LA PAGE HELLOWORLD

La page helloworld accessible à l'URL *<webapp>/jsp/site/Portal.jsp?page=helloworld* affiche le message "Bonjour Monde!" si votre navigateur est en Français, ou "Hello World!" sinon.

21. CONFIGURATION LUTECE AVANCÉE

CONFIGURATION DE L'APPLICATION

Les fichiers de configuration de l'application se trouvent dans le répertoire WEB-INF/conf de la webapp. Les fichiers à configurer sont les suivants :

- `config.properties`
- `db.properties`

CONFIGURATION DU FICHIER CONFIG.PROPERTIES

Ce fichier est à mettre à jour au moment du déploiement de l'application :

Lors du premier lancement de l'application, autoInit est à true pour vous permettre la mise à jour automatique des chemins absolus du fichier. Par la suite, il est positionné à false.

```
# AutoInit for first deployment
autoInit=false
```

Renseigner ici les informations nécessaires pour l'envoi de mail (notamment l'adresse IP du serveur SMTP).

```
 # Mail sending parameters
 mail.server=
 mail.list.separator=;
 mail.type.plain=text/plain;charset=
 mail.type.html=text/html;charset=
 mail.noreply.email=noreply@nowhere.org
 mail.charset=utf-8
```

Renseigner ici les chemins complets pour accéder aux fichiers properties lutece, jtidy et webmaster

```
 # Properties files
 file.lutece.properties=C:/tomcat/webapps/lutece/WEB-INF/conf/lutece.properties
 file.jtidy.properties=C:/tomcat/webapps/lutece/WEB-INF/conf/jtidy.properties
 file.webmaster.properties=C:/tomcat/webapps/lutece/WEB-INF/conf/webmaster.properties
 file.dir.plugins=C:/tomcat/webapps/lutece/WEB-INF/conf/plugins
```

Renseigner ici les chemins complets des fichiers de log (ces champs sont automatiquement mis à jour quand la propriété `autoInit=true`)

```
# Logs files
 file.lutece.error.log=C:/tomcat/webapps/lutece/WEB-INF/logs/errors.log
 file.lutece.app.log=C:/tomcat/webapps/lutece/WEB-INF/logs/application.log
 file.pool.log=C:/tomcat/webapps/lutece/WEB-INF/logs/pool.log
```

Si error.page.debug=true, le détail de l'erreur s'affiche dans la page. Si error.page.debug=false, le message défini dans error.page.message s'affiche dans la page.

```
# Error page management
 error.page.debug=true
 error.page.message=Veuillez contacter immédiatement l'administrateur de l'application
```

Activer ou non la gestion du cache pour les templates, le contenu des pages ou les articles. En production, il est recommandé de mettre ces valeurs à `true`.

```
 # Caches management
 service.templates.cache.enable=false
 service.pages.cache.enable=false
 service.articles.cache.enable=false
```

Renseigner ici les chemins d'accès aux fichiers de log : application.log, error.log et pool.log.

```
 # log4j parameters
 log4j.rootLogger=WARNING, Error
 log4j.logger.lutece.error=ERROR, Error, Console
 log4j.logger.lutece.pool=ERROR, Error
 ...
 log4j.appender.Error.File=C:/tomcat/webapps/lutece/WEB-INF/logs/error.log
 log4j.appender.Application.File=C:/tomcat/webapps/lutece/WEB-INF/logs/application.log
 ...
```

MISE À JOUR DU FICHIER DB.PROPERTIES

Choix du service de connexion :

- **LuteceConnectionService** pour utiliser le pool de connexion de l'application Lutèce.
- **TomcatConnectionService** pour utiliser le pool de connexion de Tomcat.

```
 portal.poolservice=fr.paris.lutece.util.pool.service.LuteceConnectionService
```

OU

```
portal.poolservice=fr.paris.lutece.util.pool.service.TomcatConnectionService
```

Renseigner ici le nom du pilote à charger pour la connection à une base MySQL.

```
portal.driver=org.gjt.mm.mysql.Driver
```

Renseigner portal.url pour identifier la source des données JDBC de la base Lutece. portal.user et portal.password renseignent le code et le mot de passe pour que l'application se connecte à la base lutece.

```
portal.url=jdbc:mysql://127.0.0.1
 /lutece?autoReconnect=true&useUnicode=yes&
 characterEncoding=utf8
 portal.user=lutece
 portal.password=lutece
```

Renseigner ici le nombre de connexions que le pool doit créer lors de son initialisation.

```
portal.initconns=2
```

Renseigner ici le nombre maximal de connexions simultanées pour le pool portal.

```
portal.maxconns=20
```

Renseigner ici le temps maximum d'attente lors d'une connexion à la base de données (en millisecondes).

```
portal.logintimeout=2
```

Requête permettant de tester la validité de la connexion. La valeur varie selon la base de données concernée :

- `SELECT 1 pour une base de type MySQL`
- `SELECT SYSDATE FROM dual pour une base Oracle.`

```
portal.checkvalidconnectionsql=SELECT 1
```

Remarque : si vous modifiez un des fichiers `properties`, pensez à arrêter tomcat et à le relancer pour que les nouveaux paramètres soient pris en compte.

Le fichier webmaster.properties est également paramétrable. Une interface permet de le faire simplement depuis le module

d'administration.

22. INITIALISATION DE LA BASE DE DONNÉES

L'initialisation de la base de données peut être effectuée après assemblage de la webapp. Le système d'initialisation utilise :

- des fichiers .sql provenant du coeur et des plugins (dans leurs dossiers src/sql/*)
- Ant avec le script de construction "target/<webapp>/WEB-INF/sql/build.xml"

Lors de l'assemblage de la webapp, les fichiers .sql sont tous rassemblés dans les sous-dossiers de "target/<webapp>WEB-INF/sql". Ant execute en premier les fichiers de création et d'initialisation des tables du coeur de Lutèce (`create_db_lutece_core.sql` et `init_db_lutece_core.sql`). Ensuite, Ant exécute les fichiers SQL de création et d'initialisation des plugins. L'ordre d'exécution des scripts dépend du nom des fichiers .sql, ce qui permet de correctement ordonner les scripts qui respectent les conventions de nommage.

Ant utilise le fichier db.properties, qui doit donc déjà contenir les informations de connexion au serveur de base de données (nom d'utilisateur, mot de passe, nom de la base de données à créer). Ant commence par vider totalement la base de données utilisée.

Remarque : si un plugin nécessite un pool de connexion particulier, il suffit de définir ce pool dans le fichier `db.properties`. Ce pool doit avoir le même nom que le plugin qui l'utilisera.

GUIDES DE DÉVELOPPEMENT DE COMPOSANTS

23. DÉVELOPPER LE SQUELETTE DU PLUGIN AVEC PLUGINWIZARD
24. CRÉER UNE APPLICATION FRONT OFFICE (XPAGE)
25. CRÉER MA PREMIÈRE APPLICATION FRONT OFFICE : MON PLUGIN HELLOWORLD
26. CRÉER UN PORTLET FRONT OFFICE
27. CRÉER UN BLOC DYNAMIQUE DE PAGE (PAGEINCLUDE)
28. CRÉER UN SERVICE DE FOURNITURE DE CONTENU (CONTENTSERVICE)
29. DÉFINIR DES FILTRES ET SERVLETS DE PLUGIN
30. CRÉER UNE FONCTIONNALITÉ BACK OFFICE (ADMINFEATURE)
31. CRÉER UN BLOC POUR LE TABLEAU DE BORD BACK-OFFICE (DASHBOARDCOMPONENT)
32. CRÉER UN SERVICE D'EXTENSION DE L'ÉDITEUR DE TEXTE RICHE (HTMLSERVICE OU LINKSERVICE)
33. CRÉER UN TRAITEMENT D'ARRIÈRE-PLAN (DAEMON)
34. CRÉER UNE TÂCHE DE WORKFLOW

23. DÉVELOPPER LE SQUELETTE DU PLUGIN AVEC PLUGINWIZARD

PluginWizard est un plugin Lutece permettant de générer les principaux fichiers d'un plugin Lutece :

- Les classes Java de la couche métier
- Les classes Java de la couche présentation
- Les classes Java des XPage
- Les JSP de l'interface d'administration
- Les templates HTML
- Les fichiers ressources d'internationalisation (i18n)
- Le fichier properties de configuration du plugin
- Le fichier de context de Spring
- Le fichier XML de définition du plugin
- Les scripts SQL de création et d'initialisation de la base de données
- Le fichier pom.xml pour construire le projet avec Maven
- Les fichiers de documentation au format xDoc

Cet assistant de génération débute par la saisie du nom du plugin. Si une génération a déjà été réalisée pour un plugin du même nom, l'application propose de récupérer ou de réinitialiser complètement les anciennes données de génération.

Création
Description
Métier
Administration
XPage
Portlet
REST
Génération

Cet assistant permet de générer un plugin Lutèce. A chaque étape vous pourrez décrire les composants à générer. A la fin de celles-ci pluginwizard vous délivrera une archive avec tous les fichiers sources et de configuration nécessaires à votre plugin.

Créer un plugin

Nom du plugin :

En minuscules et composé uniquement de caractères appartenant à l'ensemble [a..z]

Cet assistant va générer les composants suivants :

- Les classes Java de la couche métier
- Les classes Java de la couche présentation
- Les classes Java des XPage
- Les JSP de l'interface d'administration
- Les templates HTML
- Les fichiers ressources d'internationalisation (i18n)
- Le fichier properties de configuration du plugin
- Le fichier de context de Spring
- Le fichier XML de définition du plugin
- Les scripts SQL de création et d'initialisation de la base de données
- Le fichier pom.xml pour construire le projet avec Maven

Valider

L'utilisateur est ensuite guidé dans l'avancée des étapes de définition des paramètres de génération. Il peut à tout moment revenir à l'étape précédente.

La dernière étape affiche le récapitulatif et propose de télécharger un fichier archive (zip) contenant l'ensemble des fichiers générés.

Voici la liste des étapes :

- **Description** : saisie des paramètres généraux du plugin (description, auteur, version, ...)
- **Métier** : permet de définir un ou plusieurs objets métiers
- **Administration** : permet de créer une ou plusieurs AdminFeature. Si ces AdminFeatures sont associées à des objets métiers, la fonctionnalité générée correspondra à la gestion de base de l'objet métier (liste, création, modification et suppression).
- **XPage** : permet de créer une ou plusieurs XPages. Si ces XPages sont associées à des objets métiers, la fonctionnalité générée correspondra à la gestion de base de l'objet métier (liste, création, modification et suppression).
- **Portlet** : saisie des paramètres généraux du plugin (description, auteur, version, ...)
- **REST** : propose la création de Web Services REST exposant les fonctions de création, modification et suppression d'un objet métier
- **Génération** : propose la génération des fichiers, selon éventuellement plusieurs modèles de génération, ainsi que le téléchargement du résultat dans un fichier zip.

Ce plugin peut être testé sur le site de démonstration de Lutece à l'adresse suivante :

http://dev.lutece.paris.fr/site-demo/jsp/site/Portal.jsp?page=pluginwizard

Il est néanmoins préférable de travailler sur une instance dédiée ou locale de manière à garder les informations de génération.

NB : PluginWizard offre la possibilité d'ajouter ou de personnaliser des générateurs. Il permet également de définir plusieurs modèles de génération, donc il ne faut pas hésiter à l'adapter pour qu'il réponde à des objectifs précis.

Dans la suite du livre, les composants pouvant être générés par PluginWizard seront indiqués par le macaron suivant :

24. CRÉER UNE APPLICATION FRONT OFFICE (XPAGE)

Une XPage est une page spéciale qui ne fait pas partie de l'arborescence des pages construites avec des portlets. Elle est autonome et s'appelle en utilisant le parametre **page** comme suit :

```
Portal.jsp?page=ma xpage
```

Ce composant est typiquement destiné à réaliser des applications fonctionnant au sein du portail.

IMPLÉMENTATION DE BASE

La création d'un XPage nécessite seulement deux actions :

- la création d'une classe Java qui implemente l'interface XPageApplication
- la déclaration de la XPage dans le fichier xml du plugin

IMPLÉMENTATION DE L'INTERFACE XPAGEAPPLICATION

L'implementation de l'interface XPageApplication ne nécessite que de définir la seule méthode suivante:

```
XPage getPage( HttpServletRequest request, int nMode, Plugin plugin)
```

Dans le chapitre suivant, plusieurs implémentations sont présentées dans le cadre d'un plugin **HelloWorld**.

DÉCLARATION DANS LE FICHIER XML DU PLUGIN

La déclaration doit se faire comme suit :

```
<applications>
 <application>
 <application-id>mypage</application-id>
 <application-class>fr.paris.lutece.plugins.myplugin.web.MyApp </application-class>
 </application>
<applications>
```

PRINCIPE DE MISE EN OEUVRE DU MODÈLE MVC

La framework MVC, évoqué précédemment dans le livre, propose une implémentation MVCApplication de l'interface XPageApplication offrant des mécanismes permettant de découper les fonctions selon le pattern Modèle-Vue-Controleur.

Le point d'entrée de la XPage joue le rôle de contrôleur. C'est la classe MVCApplication qui prend en charge cette fonction en implémentant la méthode getPage de l'interface.

Les actions et les vues sont désignées par des annotations `@Action` et `@View`.

Le contrôleur dispatche les traitements sur les méthodes annotées en fonction des paramètres de la **request**.

L'annotation `@Controller` à définir sur la classe dérivée de **MVCApplication** sert à définir des paramètres de la classe.

La page d'accueil de la XPage doit positionner le paramètre **defaultView** de l'annotation `@View` à `true`.

LES VUES

Le framework MVC apporte une notion de vues et d'actions.

Les vues servent à afficher un contenu à l'utilisateur. Une vue est une méthode d'un contrôleur ayant l'annotation `@View`. Cette annotation prend en paramètre la clé unique de la vue qui sera utilisée par le framework pour la définir. Une fois le contenu d'une vue généré, elle doit renvoyer son résultat dans un objet **XPage**. La vue qui correspond à la page d'accueil de l'application doit comporter le paramètre **defaultView** de l'annotation `@View` à `true`. Lorsqu'une requête est effectuée sur une application, et qu'aucune vue ou action n'est spécifiée, alors la page d'accueil est affichée.

L'URL d'une vue est générée sur le modèle suivant :

```
jsp/site/Portal.jsp?page=nomdelapplication&view=nomdelavue
```

où :

- **nomdelapplication** est le nom de l'application (définie dans le paramètre de l'annotation **@Controller**)
- **nomdelavue** est le nom de la vue à afficher (définie comme paramètre de l'annotation **@View**).

Le mot clé **view** est donc un mot clé réservé et ne doit pas être utilisé comme nom de paramètre HTML pour une autre raison.

LES ACTIONS

Les actions servent à effectuer des traitements. Les actions ne doivent pas générer de contenu à afficher à l'utilisateur.

Une action est une méthode d'un contrôleur ayant l'annotation `@Action`. Cette annotation prend en paramètre la clé unique de l'action qui sera utilisée par le framework pour la définir. Lorsqu'une action termine son traitement, elle doit effectuer une redirection vers une vue grâce aux méthodes de redirection de la librairie (cf paragraphe suivant). Une action doit renvoyer un objet de type **XPage**. Lorsqu'elle effectue une redirection, elle doit renvoyer la **XPage** créée par la méthode de redirection.

L'URL d'une vue est générée sur le modèle suivant : jsp/site/Portal.jsp?page=nomdelapplication&action=nomdelaction

où :

- **nomdelapplication** est le nom de l'application (définie dans le paramètre de l'annotation `@Controller`)
- **nomdelaction** est le nom de l'action à effectuer (définie comme paramètre de l'annotation `@Action`).

Le mot clé action est donc un mot clé réservé et ne doit pas être utilisé comme nom de paramètre HTML pour une autre raison.

LES REDIRECTIONS

Lorsqu'une vue ou une action souhaite effectuer une redirection vers une autre vue ou une URL définie, elle doit appeler une des méthodes redirect(...) de la classe **MVCApplication**.

Ces méthodes permettent d'effectuer des redirections vers des vues de l'application en spécifiant uniquement le nom de la vue à afficher, ou vers une URL complète définie manuellement. Lorsque la redirection se fait vers une vue de l'application, des paramètres peuvent être spécifiés en les ajoutant dans une **Map**.

Une fois le traitement de la méthode de redirection terminé, la réponse HTTP est envoyé. Il n'est donc pas possible d'afficher un contenu HTML ou de faire une autre redirection pour ignorer la première.

Les méthodes de redirection renvoient des objets de type **XPage**. Ces objets doivent être renvoyés par les vues ou actions ayant appelé la méthode de redirection.

EXEMPLE DE CODE

Voici le code d'une XPage proposant 2 vues :

- la page d'accueil affichant une liste de contacts
- la page de création d'un contact

ainsi que 2 actions :

- créer un contact
- supprimer un contact

```
// MVCDemoApp.java

@Controller( xpageName = "mvcdemo" , pageTitle = "mvcdemo.pageTitle" , pagePath = "mvcdemo.pagePathLabel" )
public class MVCDemoApp extends MVCApplication
{

    private static final String TEMPLATE_HOME = "/skin/plugins/mvcdemo/home.html";
    private static final String TEMPLATE_FORM = "/skin/plugins/mvcdemo/form.html";
    private static final String VIEW_HOME = "home";
    private static final String VIEW_ADD_CONTACT = "addContact";
    private static final String ACTION_ADD_CONTACT = "addContact";
    private static final String ACTION_REMOVE_CONTACT = "removeContact";
    private static final String MARK_CONTACTS_LIST = "contacts_list";
    private static final String PARAMETER_ID = "id";


    @View( value = VIEW_HOME , defaultView = true )
    public XPage viewHome(HttpServletRequest request)
    {
        Map<String,Object> model = getModel();
        model.put( MARK_CONTACTS_LIST , ContactService.getContacts() );
        return getXPage( TEMPLATE_HOME, request.getLocale(  ) , model );
    }

    @View( VIEW_ADD_CONTACT )
    public XPage viewAddContact(HttpServletRequest request)
    {
        return getXPage( TEMPLATE_FORM, request.getLocale(  ) );
    }

    @Action( ACTION_ADD_CONTACT )
    public XPage addContact(HttpServletRequest request)
    {
        Contact contact = new Contact();
        populate(contact, request);

        if ( !validateBean(contact))
        {
            return redirectView( request , VIEW_ADD_CONTACT );
        }
        ContactService.addContact(contact);
        return redirectView( request , VIEW_HOME );
    }

    @Action( ACTION_REMOVE_CONTACT )
    public XPage removeContact(HttpServletRequest request)
    {
        String strContact = request.getParameter( PARAMETER_ID );
        ContactService.removeContact( strContact );
        return redirectView( request , VIEW_HOME );
    }
```

FONCTIONS AVANCÉES

FORMULAIRES MULTI-ACTIONS

Exemple de formulaire mono action :

```
<form action="jsp/site/Portal.jsp" >
      <input type="hidden" name="page" value="myplugin" />
      <input type="hidden" name="action" value="action1" />
      ...
      <button type="submit" value="Action 1" />
</form>
```

Exemple de formulaire multi-actions :

```
<form action="jsp/site/Portal.jsp">
 <input type="hidden" name="page" value="myplugin" />
 ...
 <button type="submit" name=action_action1 value="Action 1" />
 <button type="submit" name=action_action2 value="Action 2" />
 <button type="submit" name=view_view1 value="Action 3" />
</form>
```

Les prefix **action_** et **view_** dans les noms des boutons sont interprétés comme des appels d'actions ou de vues. L'autre partie du nom est interprétée comme la valeur de l'action ou de la vue.

RENVOIE DE FLUX JSON OU XML PAR LA XPAGE

A partir de Lutece 5.0

Cette fonctionnalité peut être utile pour réaliser certaine partie de la page en **Ajax** tout en gardant le code de la partie serveur dans la même classe.

Pour cela deux méthodes ont été ajoutées au framework dans la classe **MVCApplication** :

- XPage responseJSON(HttpServletRequest request)
- XPage responseXML(HttpServletRequest request)

Voici un exemple qui renvoie une liste d'images au format JSON :

```
@View( VIEW_LIST_IMAGES )
  public XPage viewListImages( HttpServletRequest request )
  {
        String strTopicId = request.getParameter( Constants.PARAMETER_TOPIC_ID );
        JSONArray array = new JSONArray();

        if( strTopicId != null )
        {
            int nTopicId = Integer.parseInt(strTopicId);
            List<Image> list = ImageHome.findByTopic( nTopicId , _plugin );
            for( Image image : list )
            {
                JSONObject jsonImage = new JSONObject();
                jsonImage.accumulate( "id", image.getId() );
                jsonImage.accumulate( "name", image.getName() );
                array.add( jsonImage );
            }
        }
        return responseJSON( array.toString() );

  }
```

25. CRÉER MA PREMIÈRE APPLICATION FRONT OFFICE : MON PLUGIN HELLOWORLD

Ce chapitre décrit la création de plusieurs XPageApplications, utilisant progressivement de plus en plus de fonctionnalités de Lutece. La dernière version correspond à la XPage utilisée pour tester l'environnement de développement utilisée dans un des chapitres précédents.

PREMIÈRE IMPLÉMENTATION

Voici comment réaliser le plugin le plus simple. L'objet de celui-ci est d'afficher le texte "Hello, world!". Nous allons utiliser pour cela une XPageApplication. Le nom du plugin sera **helloworld** et celui de la classe d'implémentation de l'application sera `HelloWorldApp` . En respectant les règles de nommage et de structure des plugins, la classe sera créée dans le package **fr.paris.lutece.plugins.helloworld.web** .

Voici l'implémentation la plus simple de cette classe.

```
package fr.paris.lutece.plugins.helloworld.web;

import javax.servlet.http.HttpServletRequest;

import fr.paris.lutece.portal.service.plugin.Plugin;
import fr.paris.lutece.portal.web.xpages.XPage;
import fr.paris.lutece.portal.web.xpages.XPageApplication;

public class HelloWorldApp implements XPageApplication
{
     public XPage getPage( HttpServletRequest request, int nMode, Plugin plugin )
     {
          XPage page = new XPage( );
          page.setContent( "Hello, World!" );
          page.setTitle( "Hello, World!" );
          page.setPathLabel( "Hello, World!" );
          return page;
     }
}
```

Dans cette première implémentation, nous créons notre page en instanciant un nouvel objet XPage pour lequel nous définissons les attributs suivants :

Attribut	Description
content	Le contenu de la page
title	Le titre de la page (s'affiche dans la barre de titre du navigateur)
pathLabel	Le nom de la page dans le fil d'ariane du portail

Cette implémentation illustre le fonctionnement de base d'une application XPage, néanmoins elle ne respecte pas certaines règles élémentaires de développement qui veulent que des éléments variables tels que des libellés ou du code HTML ne soient pas écrits "en dur" dans le code Java. Pour corriger cela, nous allons réaliser une nouvelle implémentation utilisant certains services de base de Lutèce.

PACKAGER LE PLUGIN

Pour réaliser un plugin opérationnel, il faut à présent créer son fichier de déploiement. Celui doit être créé dans le répertoire **WEB-INF/plugins** et s'appelera **helloworld.xml** .

Voici le contenu du fichier :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<plug-in>
     <!-- Plugin Informations -->
     <name>helloworld</name>
     <class>fr.paris.lutece.portal.service.plugin.PluginDefaultImplementation</class>
     <version>1.0</version>
     <documentation></documentation>
     <installation></installation>
     <changes></changes>
     <user-guide></user-guide>
     <description>Exemple d'application</description>
     <provider>Mairie de Paris</provider>
     <provider-url>http://lutece.paris.fr</provider-url>
     <icon-url>images/admin/skin/plugins/helloworld/helloworld.png</icon-url>
     <copyright>Copyright 2001-2014 Mairie de Paris</copyright>
     <db-pool-required>0</db-pool-required>

     <!-- Xpage configuration -->
     <applications>
         <application>
        <application-id>helloworld</application-id>
           <application-class>fr.paris.lutece.plugins.helloworld.web.HelloWorldApp</application-class>
         </application>
     <applications>
</plug-in>
```

Il faut également créer une icône pour l'application et l'enregistrer dans le répertoire **images/admin/skin/plugins/helloworld** .

DEUXIÈME IMPLÉMENTATION

Cette deuxième implémentation va déporter le code HTML du contenu de la page dans un fichier template et les libellés du title et du PathLabel dans le fichier de propriété du plugin.

```
package fr.paris.lutece.plugins.helloworld.web;

import javax.servlet.http.HttpServletRequest;

import fr.paris.lutece.portal.service.plugin.Plugin;
import fr.paris.lutece.portal.service.template.AppTemplateService;
import fr.paris.lutece.portal.service.util.AppPropertiesService;
import fr.paris.lutece.portal.web.xpages.XPage;
import fr.paris.lutece.portal.web.xpages.XPageApplication;
import fr.paris.lutece.util.html.HtmlTemplate;

public class HelloWorldApp2 implements XPageApplication
{
     private static final String TEMPLATE_HELLO_WORLD = "site/plugins/helloworld/helloworld.html";
     private static final String PROPERTY_PAGE_TITLE = "helloworld.pageTitle";
     private static final String PROPERTY_PAGE_PATH_LABEL = "helloworld.pagePathLabel";

     public XPage getPage( HttpServletRequest request, int nMode, Plugin plugin )
     {
          XPage page = new XPage( );

          HtmlTemplate template = AppTemplateService.getTemplate( TEMPLATE_HELLO_WORLD );
          String strPageTitle = AppPropertiesService.getProperty( PROPERTY_PAGE_TITLE );
          String strPagePathLabel = AppPropertiesService.getProperty( PROPERTY_PAGE_PATH_LABEL );

          page.setContent( template.getHtml() );
          page.setTitle( strPageTitle );
          page.setPathLabel( strPagePathLabel );
```

```
            return page;
        }
}
```

cette implémentation nécessite la création d'un fichier HTML **helloworld.html** dans le répertoire **WEB-INF/templates/skin/plugins/helloworld**.

Voici le contenu de fichier :

```
<h1>Hello, World!</h1>
```

Il faut également créer un fichier **helloworld.properties** dans le répertoire **WEB-INF/conf/plugins** .

Voici le contenu de ce fichier :

```
# fichier de configuration du plugin helloworld

helloworld.pageTitle=Application Hello World
helloworld.pagePathLabel=Hello World
```

TROISIÈME IMPLÉMENTATION (MULTILINGUE)

Voici une dernière implémentation qui permettrait de proposer facilement l'application en plusieurs langues. Pour cela, Lutece utilise les fonctions d'internationalisation de Java à travers le service **I18nService** .

Pour gérer les resources en plusieurs langues il faut créer, dans le package **fr.paris.lutece.plugins.helloworld.resources** , les fichiers properties correspondants à chaque localisation.

- **helloworld_messages.properties** - langage par défaut
- **helloworld_messages_fr.properties** - français
- **helloworld_messages_fr_FR.properties** - français (France)
- **helloworld_messages_fr_CA.properties** - français (Canada)
- **helloworld_messages_en_US.properties** - anglais (Etats-Unis)
- **...**

Voici le contenu du fichier **helloworld_messages_fr.properties** :

```
 # fichier de ressources du plugin helloworld en français

content=Bonjour le Monde !
pageTitle=Application Bonjour le Monde
pagePathLabel=Bonjour le Monde
```

Note Le fichier de configuration **helloworld.properties** ne sera plus nécessaire dans cet exemple.

Voici le nouveau contenu du fichier **heloworld.html** :

```
<h1>#i18n{ helloworld.content }</h1>
```

Enfin l'implémentation de l'application sera modifiée comme suit :

```
package fr.paris.lutece.plugins.helloworld.web;

import java.util.Locale;

import javax.servlet.http.HttpServletRequest;

import fr.paris.lutece.portal.service.i18n.I18nService;
import fr.paris.lutece.portal.service.plugin.Plugin;
import fr.paris.lutece.portal.service.template.AppTemplateService;
import fr.paris.lutece.portal.web.xpages.XPage;
import fr.paris.lutece.portal.web.xpages.XPageApplication;
import fr.paris.lutece.util.html.HtmlTemplate;

public class HelloWorldApp3 implements XPageApplication
{
      private static final String TEMPLATE_HELLO_WORLD = "site/plugins/helloworld/helloworld.html";
      private static final String PROPERTY_PAGE_TITLE = "helloworld.pageTitle";
      private static final String PROPERTY_PAGE_PATH_LABEL = "helloworld.pagePathLabel";

      public XPage getPage( HttpServletRequest request, int nMode, Plugin plugin )
      {
            XPage page = new XPage( );

            Locale locale = request.getLocale();
            HtmlTemplate template = AppTemplateService.getTemplate( TEMPLATE_HELLO_WORLD , locale );
            String strPageTitle = I18nService.getLocalizedString( PROPERTY_PAGE_TITLE , locale );
            String strPagePathLabel = I18nService.getLocalizedString( PROPERTY_PAGE_PATH_LABEL , locale );

            page.setContent( template.getHtml() );
            page.setTitle( strPageTitle );
            page.setPathLabel( strPagePathLabel );

            return page;
      }
}
```

QUATRIÈME IMPLÉMENTATION (MULTILINGUE + MVC)

Cette dernière implémentation utilise le framework MVC disponible à partir de la version Lutece 4.1

```
package fr.paris.lutece.plugins.test.web;

import fr.paris.lutece.portal.web.xpages.XPage;
import fr.paris.lutece.portal.util.mvc.xpage.MVCApplication;
import fr.paris.lutece.portal.util.mvc.commons.annotations.View;
import fr.paris.lutece.portal.util.mvc.xpage.annotations.Controller;
import javax.servlet.http.HttpServletRequest;

@Controller( xpageName = "helloworld" , pageTitleI18nKey = "helloworld.pageTitle" , pagePathI18nKey =
"helloworld.pagePathLabel" )
public class HelloWorldApp4 extends MVCApplication
{
     // Templates
     private static final String TEMPLATE_HELLO_WORLD = "site/plugins/helloworld/helloworld.html";
     private static final String VIEW_HELLO_WORLD = "helloWorld";

     @View( value = VIEW_HELLO_WORLD, defaultView = true )
     public XPage viewHelloWorld( HttpServletRequest request )
     {
         return getXPage( TEMPLATE_HELLO_WORLD, request.getLocale(  ));
     }
}
```

26. CRÉER UN PORTLET FRONT OFFICE

Les pages de Lutèce sont composées de portlets. Il est possible de définir de nouveaux types de portlets qui seront alors disponibles pour la composition des pages et permettront d'ajouter de nouvelles fonctions. Les nouveaux portlets doivent être "packagés", avec leurs éventuelles fonctions d'administration associées, sous la forme d'un plugin.

Ce document a pour but de décrire les étapes du processus de création d'un nouveau type de portlet. Les modifications à prévoir concernent l'ensemble de l'application : java, base de données, templates html, xsl, jsp.

On adoptera par convention dans l'ensemble du document la notation **<type>** pour indiquer le nom du nouveau type de portlet.

LES CLASSES À DÉFINIR EN JAVA

En java, l'ajout d'un nouveau type de portlet implique la création de trois classes métiers : Portlet.java, PortletHome.java, PortletDAO.java appartenant au package **fr.paris.lutece.portal.business.portlet** pour la partie Back Office et un bean servant à l'affichage Web : PortletJspBean.java appartenant au package **fr.paris.lutece.plugins.<myplugin>.web.portlet** dont suit le descriptif :

<TYPE>PORTLET.JAVA

Elle doit dériver de la classe abstraite : **Portlet** qui elle-même implémente l'interface : **XmlContent** (package **fr.paris.lutece.plugins.<myplugin>.business**).

```
public class Portlet extends Portlet
{
   ...
}
```

Cet héritage impose à la classe <type>Portlet l'implémentation de deux méthodes :

- public String **getXml()**;
- public String **getXmlDocument()**; qui retournent le contenu du portlet au format xml.

De même, elle hérite des accesseurs (méthodes set et get) sur les propriétés caractérisant un portlet, regroupés dans la classe Portlet, ainsi que des constantes (tags) définis dans l'interface **XmlContent** .

Enfin, il faut également prévoir deux autres méthodes nécessaires pour la mise à jour du portlet :

- public void **update()**
- public void **remove()** qui appellent les méthodes de mise à jour de la classe DAO associée ainsi que les différents accesseurs qui caractérisent le nouveau type de portlet.

A partir de Lutece v4, il est possible de réaliser des portlets qui produisent directement le contenu HTML sans passer par XML/XSLT. Ces portlets ne gèrent pas de changement style.

La methode à implémenter est :

- public String **getHtmlContent**();

<TYPE>PORTLETHOME.JAVA

Elle doit dériver de la classe abstraite : **PortletHome** qui elle-même implémente l'interface **PortletHomeInterface** .

```
public class MyPortletHome extends PortletHome
{
   ...
}
```

Elle doit donc implémenter les deux méthodes :

- public InterfaceDAO **getDAO()** : renvoie l'instance unique de la classe PortletDAO décrite ci-dessous
- public int **getPortletTypeId()** : renvoie l'identifiant du type de portlet stocké dans la base

Celle classe devant fonctionner comme un singleton, la méthode :

- public static PortletHome **getInstance()** est nécessaire pour accéder à l'instance unique.

<TYPE>PORTLETDAO.JAVA

Elle doit implémenter l'interface : **InterfaceDAO**.

```
public class PortletDAO implements InterfaceDAO
{
   ...
}
```

Elle doit donc implémenter les méthodes suivantes d'accès aux données :

- public void **insert**(Portlet portlet) throws AppException;
- public void **delete**(int nPortletId) throws AppException;
- public void **store**(Portlet portlet) throws AppException;
- public Portlet **load**(int nPortletId) throws AppException;

Outre ces méthodes indispensables, le développeur est libre d'ajouter dans cette classe autant de méthodes d'accès aux données qu'il désire et qui lui sont nécessaires pour la gestion du nouveau portlet.

<TYPE>PORTLETJSPBEAN.JAVA

Elle doit dériver de la classe : **PortletJspBean** qui elle-même implémente l'interface **PortletJspBeanInterface** .

```
public class MyPortletJspBean extends PortletJspBean
{
   ...
}
```

Ceci implique l'implémentation des méthodes suivantes pour la gestion des formulaires de saisie d'un portlet :

- public String **getCreate**(HttpServletRequest request); : Affichage du formulaire de création du portlet
- public String **doCreate**(HttpServletRequest request); : Traitement de création du portlet
- public String **getModify**(HttpServletRequest request); : Affichage du formulaire de modification du portlet
- public String **doModify**(HttpServletRequest request); : Traitement de modification du portlet (doit appeler la méthode

update de la classe <type>Portlet)
- public String **getPropertiesPrefix**(); : Renvoie le préfixe des propriétés du portlet définies dans le fichier de configuration *lutece.properties*

Ces différentes méthodes seront appelées par les jsp en charge de la création et de la modification d'un portlet. Ces dernières sont décrites ci-dessous.

De plus, **<type>PortletJspBean** hérite des nombreuses méthodes de la classe **PortletJspBean** permettant de récupérer des propriétés sur les formulaires à afficher.

Enfin, cette classe peut accueillir la définition d'autres méthodes nécessaires à des portlets qui ont besoin de traitements supplémentaires (par exemple, le traitement de l'enregistrement d'un fichier dans un portlet de type Téléchargement : DownloadFilePortlet).

Il est à noter que le traitement de la suppression d’un nouveau portlet n’a pas besoin d’être géré par le développeur car cela est pris en charge par le bean **AdminJspBean** qui appelle la méthode delete redéfinit dans **<type>Portlet** .

Le traitement est donc analogue quelque soit le type de portlet.

JSP

Les jsp indispensables à la gestion du nouveau portlet sont au nombre de 4 :

- ***CreatePortlet<type>.jsp***
- ***DoCreatePortlet<type>.jsp***
- ***ModifyPortlet<type>.jsp***
- ***DoModifyPortlet<type>.jsp***

Ils font appel aux méthodes de la classe **<type>PortletJspBean** décrites précédemment :

JSP	Méthode de la classe <type>PortletJspBean
CreatePortlet<type>.jsp	public String getCreate(HttpServletRequest request)
DoCreatePortlet<type>.jsp	public String doCreate(HttpServletRequest request);
ModifyPortlet<type>.jsp	public String getModify(HttpServletRequest request);
DoModifyPortlet<type>.jsp	public String doModify(HttpServletRequest request);

Ces JSP doivent également gérer l'authentification de l'administration.

Exemple de JSP d'affichage : CreatePortlet<Type>.jsp

```
<%@ include file="../PluginAdminHeader.jsp" %>
<jsp:useBean id="<type>Portlet" scope="session" class="fr.paris.lutece.portal.web.portlet.Portlet<Type>JspBean" />
<jsp:useBean id="user" scope="session" class="fr.paris.lutece.portal.web.user.UserJspBean" />
<% if( user.check( "DEF_ADMIN" ) ) { %>
 <%= <type>Portlet.getCreate( request ); >
<% } else { response.sendRedirect( user.getUrlAccesRefuse() ); } %>
<%@ include file="../../AdminFooter.jsp" %>
```

Exemple de JSP de traitement : DoCreatePortlet<Type>

```
<%@ include file="../PluginAdminHeader.jsp" %>
<jsp:useBean id="<type>Portlet" scope="session" class="mdp.portail.web.rubrique.Portlet<Type>JspBean" />
<jsp:useBean id="user" scope="session" class="mdp.portail.web.user.UserJspBean" />
<% if( user.check( "DEF_ADMIN" ) ) { response.sendRedirect( <type>Portlet.doCreate( request ) ); }
 else { response.sendRedirect( user.getAccessDeniedUrl() ); } %>
<%@ include file="../../AdminFooter.jsp" %>
```

LES AUTRES FICHIERS

FEUILLES DE STYLE XSL

Plusieurs feuilles de style sont à développer pour la gestion de l'affichage du nouveau type de portlet.

Une première catégorie concerne l’affichage du portlet sur le site. Par convention, son nom est :

portlet_<type>_<suffixe>.xsl où <suffixe> caractérise la spécification de la feuille de style (**par exemple :** portlet_html_no_title.xsl).

La seconde catégorie sert à afficher le portlet en mode administration du site. Il faut que chaque feuille de style de la première catégorie ait une feuille de style correspondante dans l'administration. En effet, cette feuille de style est utilisée pour la prévisualisation de la rubrique dans le module d'administration. Par convention, son nom est :

portlet_<type>_<suffixe>_adm.xsl . C’est une copie de **portlet_<type>_<suffixe>.xsl** à laquelle il faut rajouter les boutons ‘Modifier cette rubrique’, ‘Supprimer cette rubrique’.

Toutes ces feuilles de style devront être stockées en base depuis l'interface d'administration (cf paragraphe Base de données)

TEMPLATES HTML

Les formulaires HTML de saisie d’un nouveau portlet et de modification sont découpés en plusieurs fichiers :

- create_portlet.html et modify_portlet.html : contiennent respectivement le code commun aux formulaires de saisie et de modification des différentes portlets

- des templates propres au portlet concerné qui viennent compléter le formulaire de saisie en ajoutant des champs et des traitements javascript qui leur sont spécifiques. Ces templates sont à rajouter pour chaque nouveau type de portlet dans le répertoire :
  - WEB-INF/templates/admin/plugins/<type>/ **et doivent être spécifié dans le fichier properties du plugin :**

Propriété	Description
portlet.<type>.create.title	Titre du formulaire d'ajout de rubrique du type <type>
portlet.<type>.create.url	Url de la jsp qui prend en charge l'ajout d'une rubrique de type <type>
portlet.<type>.create.script	Script javascript spécifique au formulaire d'ajout de rubrique de type <type>.
portlet.<type>.create.specific	Nom du fichier de template qui contient du code spécifique pour le type de rubrique <type>
portlet.<type>.create.specificform	Nom du fichier de template qui contient le code d'un autre formulaire à accoler au formulaire d'ajout de rubrique de type <type>
portlet.<type>.modify.title	Titre du formulaire de modification de rubrique du type <type>
portlet.<type>.modify.url	Url de la jsp qui prend en charge la modification d'une rubrique de type <type>

Seules les propriétés : **portlet.<type>.create.title** , **portlet.<type>.create.url** , **portlet.<type>.modify.title** et **portlet.<type>.modify.url** sont obligatoires.

Néanmoins il est à noter que la méthode **Submit()** en javascript doit être définie car elle est appelée sur validation du formulaire.

Pour cela, le développeur peut procéder ainsi :

- soit utiliser les scripts standards utilisés par la plupart des rubriques : **script_create_portlet.html** et **script_modify_portlet.html** (dans le répertoire *WEB-INF/templates/admin/portlet/*)
- soit définir la méthode dans le formulaire spécifique (à l'instar de la rubrique html).

BASE DE DONNÉES

Les modifications dans la base de données concernent 4 tables :

Table **portlet_type** :

Nom du champ	Description
id_portlet_type	Identifiant du type de portlet.
name	Libellé du type de portlet(c'est ce libellé qui apparaîtra dans les listes déroulantes)
url_creation	Url de la JSP qui affiche le formulaire de saisie du portlet
url_update	Url de la JSP qui affiche le formulaire de modification du portlet
home_class	Nom complet (package compris) de la classe <type>PortletHome du nouveau type de rubrique

Exemple d'enregistrement : Rubrique HTML

id_portlet_type	name	url_creation	url_update	home_class
1	Texte libre ou HTML	CreatePortletHtml.jsp	ModifyPortletHtml.jsp	fr.paris.lutece.plugins.html.business.portlet.HtmlPortletHome

Ces données sont initialisées dans la base à l'installation du plugin.

Le fichier de configuration du plugin devra donc contenir ces informations sous la forme

```
<portlet>
 <portlet-class>fr.paris.lutece.plugins.myportlet.business.portlet.MyPortletHome</portlet-class>
  <portlet-type-name>MyNew Portlet</portlet-type-name>
  <portlet-creation-url>plugins/article/CreatePortletMyPortlet.jsp</portlet-creation-url>
 <portlet-update-url>plugins/article/ModifyPortletMyPortlet.jsp</portlet-update-url>
</portlet>
```

Table **style** :

Il faut définir les styles associés à ce nouveau style de rubrique. Pour cela, il faut utiliser l'interface d'administration de Lutèce : **Gestion des Styles** .

Tables **stylesheet** et **style_mode_stylesheet** : L'interface d'administration de Lutèce **"Gestion des feuilles de style xsl"** permet de définir de nouvelles feuilles de style et de les associer aux styles stockés dans la base. Toutes les feuilles de style développées conformément au paragraphe "Feuilles de style XSL" devront être sauvegardés dans la base via cette interface.

27. CRÉER UN BLOC DYNAMIQUE DE PAGE (PAGEINCLUDE)

DÉFINITION

Les PageIncludeServices sont des services qui permettent d'inclure du code HTML dans les pages du site par le biais de bookmarks placés dans le gabarit principal du portail (**page_frameset.html**). Ces services peuvent introduire du contenu mais sont plutôt destinés à insérer du code technique à des emplacements non accessibles pour les portlets ou les XPageApp. Par exemple insérer dynamiquement, dans la partie **<head>** des pages HTML, des liens vers des feuilles de style CSS ou vers des fichiers RSS.

DÉCLARATION

Lutèce intègre par défaut plusieurs PageIncludeServices au niveau du noyau :

Nom	Description	Fournisseur
Links	Insertion des feuilles de style CSS et des Javascripts spécifiques des plugins installés	lurece-core
Métas	Insertion des valeurs des tags méta à partir de celles définies dans le fichier **webmaster.properties**	lutece-core
Themes	Gestion dynamique des chemins des feuilles de style CSS en fonction du thème associé à la page	lutece-core
Statistics	Insertion du code HTML spécifique à l'outil de mesure d'audience	lutececore
GTools	Insertion des balises Google Analytics	plugin-gtools
Piwik	Insertion des balises Piwik	plugin-piwik

Les services par défaut sont déclarés dans le fichier core.xml. Les plugins peuvent ajouter de nouveaux PageIncludeServices en les déclarant dans leur fichier XML de la manière suivante :

```
<page-include-service>
     <page-include-service-name>My Include Service</page-include-service-name>
     <page-include-service-class>fr.paris.lutece.myplugin.web.MyInclude</page-include-service-class>
</page-include-service>
```

IMPLÉMENTATION

Un PageIncludeService doit implémenter l'interface **PageInclude** .

Voici un exemple très simple d'implémentation :

```
public class MyInclude implements PageInclude
{
         private static final String MARK_MY_INCLUDE = "my_include";

         /**
         * {@inheritDoc }
         */
         @Override
         public void fillTemplate( Map<String, Object> rootModel, PageData data, int nMode, HttpServletRequest
request )
         {
               rootModel.put( MARK_MY_INCLUDE , "My include content" );
         }
}
```

28. CRÉER UN SERVICE DE FOURNITURE DE CONTENU (CONTENTSERVICE)

DÉFINITION

Un content service est un service de contenu qui a la responsabilité de produire lui-même ses pages. Chaque service de contenu dispose d'un paramètre d'appel qui permet au portail de dispatcher les requêtes sur le service. Voici quelques exemples de ContentService très utilisés dans Lutece :

Service	Paramètre	Descrition	Fournisseur
PageService	id_page	Service de construction et d'affichage des pages de Lutèce	lutece-core
XPageService	page	Conteneur d'applications XPage de Lutèce	lutece-core
SearchService	query	Service de recherche de Lutèce	lutece-core
DocumentService	id_document	Service de construction et d'affichage des documents	plugin-document

INTERFACE D'UN CONTENTSERVICE

Voici les méthodes de l'interface d'un ContentService :

Méthode	Description
getName	Renvoie le nom du service de contenu.
isInvoked	Détermine à partir des paramètres de la requête HTTP, si celle-ci lui est destinée. Par exemple, le service PageService vérifiera que la requête contient le paramètre **id_page** .
getCache	Indique si le service dispose d'un cache activé.
getCacheSize	Renvoie le nombre d'objets dans la cache.
getPage	Construit une page en fonction des paramètres de la requête et du mode.
resetCache	Vide le cache du service de contenu.

CRÉATION D'UN CONTENTSERVICE

Il est possible d'ajouter de nouveaux services de contenu par le biais de plugins. Le plugin devra définir le ContentService dans son fichier de déploiement de la manière suivante :

```
<!-- Content Service -->
  <content-services>
     <content-service>
         <content-service-id>myContentService</content-service-id>
         <content-service-class>fr.paris.lutece.plugins.myplugin.service.MyContentService</content-service-class>
     </content-service>
  </content-services>
```

29. DÉFINIR DES FILTRES ET SERVLETS DE PLUGIN

Les servlets et les filtres sont habituellement déclarés dans le fichier **WEB-INF/web.xml**. Or ce fichier est unique et ne peut en aucun cas être modifié par les plugins.

Afin d'offrir aux plugins ce type de composants, Lutece propose de créer des servlets et des filtres de plugin. Ceux-ci se déclarent dans le fichier xml.

CLASSES JAVA

Les classes Java des servlets ou des filtres sont à écrire exactement de la même manière que des servlets ou filtres standards. Seule leur déclaration est différente.

Des classes existantes, provenant par exemple de produits externes, peuvent être packagées sous forme de servlet ou filtre de plugin (exemple : le filtre de réécriture d'url utilisé par le plugin urlrewriter).

DÉCLARATION DES SERVLETS ET DES FILTRES

Les servlets et filtres sont à déclarer dans le fichier XML du plugin de la manière suivante :

```
<!-- servlets -->
     <servlets>
          <servlet>
               <servlet-name>myServlet1</servlet-name>
               <url-pattern>/servlet/plugins/myplugin/myServlet1</url-pattern>
               <servlet-class>fr.paris.lutece.plugins.myplugin.web.MyFirstServlet</servlet-class>
          </servlet>
          <servlet>
               <servlet-name>myServlet2</servlet-name>
               <url-pattern>/servlet/plugins/myplugin/myServlet2</url-pattern>
               <servlet-class>fr.paris.lutece.plugins.myplugin.web.MySecondServlet</servlet-class>
          </servlet>
     </servlets>

     <!-- filters -->
     <filters>
          <filter>
               <filter-name>myFilter</filter-name>
               <url-pattern>/*</url-pattern>
               <filter-class>fr.paris.lutece.plugins.myplugin.web.MyFilter</filter-class>
               <init-param>
                    <param-name>param1</param-name>
                    <param-value>value of param1</param-value>
               </init-param>
          </filter>
     </filters>
```

NB : Les URL des servlets doivent impérativement être basées sous **/servlet/plugins/myplugin**

30. CRÉER UNE FONCTIONNALITÉ BACK OFFICE (ADMINFEATURE)

Les fonctionnalités d'administration correspondent à l'ensemble des fonctions disponibles dans le module administrateur quelque soit son niveau d'accès : administrateur technique, webmestre ou intervenant local.

A chaque fonctionnalité correspond un droit qui peut être attribué à un utilisateur. Chaque droit est associé à un niveau pour distinguer les différents profils d'utilisateurs. Les valeurs des niveaux de droits sont les suivantes :

- 0 - Administrateur technique
- 1 - Producteur de contenu
- 2 - Webmestre
- 3 - Intervenant local

Lorsque l'utilisateur est autorisé à utiliser une fonctionnalité, elle s'affiche dans son menu d'accueil sous la forme d'un titre et d'une description. En cliquant sur le lien, l'utilisateur accède à la JSP qui est le point d'entrée de la fonctionnalité.

Cette JSP porte généralement un nom de type `Manage<MyFeature>.jsp`.

Dans le modèle **MVC**, explicité précédemment, cette JSP joue le rôle de point d'entrée du *Controller*. Comme préconisé dans Lutece, la JSP contient le minimum de code et délègue ses traitements à un **JspBean**, typiquement dénommé `<MyFeature>JspBean.java`.

Cette JSP doit ensuite être déclarée dans le fichier XML du plugin afin d'être enregistrée dans le système et apparaître dans les menus.

Voici comment tout cela se traduit en terme de code :

CODE DE LA JSP

Le code de la JSP doit vérifier que l'utilisateur courant dispose du droit nécessaire, et si c'est le cas elle appelera la méthode du JspBean associé à la fonctionnalité (ex : `getManageMyFeatureHome`) restituant la page d'accueil de celle-ci.

```
 <%@ page errorPage="../../ErrorPage.jsp" %>

<jsp:include page="../../AdminHeader.jsp" />

<jsp:useBean id="manageMyFeature" scope="session" class="fr.paris.lutece.plugins.myplugin.web.ManageMyFeatureJspBean"
/>

<% manageMyFeature.init( request, manageMyFeature.RIGHT_MANAGEMONPLUGIN ); %>
<%= manageMyFeature.getManageMyFeatureHome ( request ) %>

<%@ include file="../../AdminFooter.jsp" %>
```

CODE DU JSPBEAN

```
/**
 * This class provides the the feature MyFeature
 */
@Controller( controllerJsp = "ManageMyFeature.jsp", controllerPath = "jsp/admin/plugins/myplugin/", right =
"MYFEATURE_MANAGEMENT" )
public class MyFeatureJspBean extends MVCAdminJspBean
{
    // Templates
    private static final String TEMPLATE_MANAGE_MYFEATURE = "/admin/plugins/myplugin/manage_myfeature.html";

    // Views
    private static final String VIEW_MANAGE_MYFEATURE = "manageMyFeature";

 // Markers
    private static final String MARK_OBJECT1 = "object1";

    // Properties for page titles
    private static final String PROPERTY_PAGE_TITLE_MYFEATURE = "myplugin.manage_myfeature.pageTitle";


    /**
     * Build the Manage View
     * @param request The HTTP request
     * @return The page
     */
    @View( value = VIEW_MANAGE_MYFEATURE, defaultView = true )
    public String getManageMyFeatureHome( HttpServletRequest request )
    {
 // Get a model object that contains already default data (errors list, ...)
        Map<String, Object> model = getModel();

 // Put some new objects in the model
 model.put( MARK_OBJECT1, object1 );
 ...

        return getPage( PROPERTY_PAGE_TITLE_MYFEATURE, TEMPLATE_MANAGE_MYFEATURE, model );
    }

 ...

}
```

DÉFINITION D'UNE FONCTIONNALITÉ D'ADMINISTRATION DANS LE FICHIER DE CONFIGURATION D'UN PLUGIN

Pour installer une nouvelle fonctionnalité, celle-ci doit être "packagée" dans un plugin.

La définition de la fonctionnalité dans le fichier de configuration du plugin se fait par l'ajout des lignes suivantes :

```
<admin-feature>
   <feature-id>MYFEATURE_MANAGEMENT</feature-id>
   <feature-title>My Feature</feature-title>
   <feature-description>Description of my feature</feature-description>
   <feature-level>3</feature-level>
   <feature-url>plugins/myplugin/ManageMyFeature.jsp</feature-url>
</admin-feature>
```

31. CRÉER UN BLOC POUR LE TABLEAU DE BORD BACK-OFFICE (DASHBOARDCOMPONENT)

La page d'accueil du *back office* de Lutece permet d'afficher des blocs fonctionnels proposés par des plugins.

Ces composants s'appellent des DashboardComponents.

La réalisation de ces composants requiert :

- la création d'une classe Java qui implémente l'interface `IDashboardComponent` ou étend la classe `DashboardComponent`
- la déclaration du composant dans le fichier XML du plugin

L'organisation des blocs est gérée en base de données dans la table `core_dashboard`.

LE CODE DU COMPOSANT

```
public class MyDashboardComponent extends DashboardComponent
{
    private static final String EMPTY_STRING = "";
    private static final String TEMPLATE_MY_ADMIN_DASHBOARD = "admin/plugins/myplugin/myadmindashboard.html";

    /**
     *
     * {@inheritDoc}
     */
    @Override
    public String getDashboardData( AdminUser user, HttpServletRequest request )
    {
         String strContent;

         // build the content
         ...

         return strContent;
    }
}
```

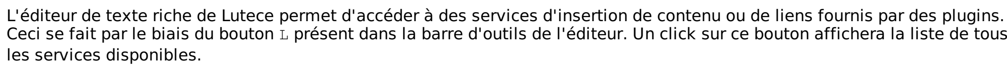
LA DÉCLARATION DU COMPOSANT DANS LE FICHIER XML DU PLUGIN

```
     <!-- Dashboard  components -->
    <dashboard-components>
        <dashboard-component>
            <dashboard-component-name>MYDASHBOARD</dashboard-component-name>
            <dashboard-component-class>fr.paris.lutece.plugins.myplugin.web.MyDashboardComponent </dashboard-
component-class>
            <dashboard-feature-right>MYFEATURE_MANAGEMENT</dashboard-feature-right>
        </dashboard-component>

        ...

    </dashboard-components>
```

32. CRÉER UN SERVICE D'EXTENSION DE L'ÉDITEUR DE TEXTE RICHE (HTMLSERVICE OU LINKSERVICE)



L'éditeur de texte riche de Lutece permet d'accéder à des services d'insertion de contenu ou de liens fournis par des plugins. Ceci se fait par le biais du bouton L présent dans la barre d'outils de l'éditeur. Un click sur ce bouton affichera la liste de tous les services disponibles.

Un plugin qui propose un service d'insertion de code HTML ou un service d'insertion de lien va devoir implémenter une interface **HtmlService** ou **LinkService** . L'arborescence des InsertService est constituée comme suit :

- une interface ancêtre généralisant la notion de InsertService et fournissant une méthode abstraite **getSelectorUI** donnant accès à une IHM de sélection de l'objet à insérer ;
- une interface **LinkService**modélisant les Services d'insertion de liens et fournissant une méthode retournant une classe de sélection implémentant
  - LinkServiceSelectionBean **;**
- une interface **HtmlService**modélisant les Services d'insertion de code HTML et fournissant une méthode retournant une classe de sélection implémentant
  - HtmlServiceSelectionBean **;**
- des classes abstraites, **DefaultLinkService** et **DefaultHtmlService**, facilitant l'implémentation des 2 interfaces ci dessus.

Note : un plugin peut proposer à la fois un LinkService et un HtmlService ;

SERVICE D'ENREGISTREMENT

Le service concernant deux fonctionnalités très similaires, nous avons donc décidé de ne créer qu'un seul service d'enregistrement : InsertServiceManager.

Des méthodes sont fournies pour récupérer séparément les HtmlService ou les LinkService ou tous les InsertService.

INTERFACE DE SÉLECTION D'UN LIEN

Comme dans la version 1.0 de Lutece, la sélection de liens ou d'objets à insérer dans un éditeur HTML se fait via une fenêtre de type popup[1].

Cela permet d'une part de ne pas perdre l'utilisateur et d'autre part de ne pas avoir à gérer la persistance des données saisies dans l'appelant (sauvegarde des données non enregistrées dans l'éditeur HTML avant l'appel de la fonction d'insertion).

INSERTION DU CODE RETOUR

L'implémentation est conditionnée par le choix précédent et la volonté de ne pas se soucier de la mémorisation des données saisies dans l'appelant.

L'insertion du code HTML correspondant à l'objet sélectionné se fait directement via une fonction javascript que doit proposer l'appelant :

insert_html (String ChaineAInsérer) .

ENREGISTREMENT DES SERVICES

L'enregistrement des implémentations de InsertService est effectué au lancement de l'application lors du chargement des plugins.

Il s'agit d'ajouter une section <html-service> (link-service) dans le fichier de configuration du plugin :

```
<!-- Links Service -->
<link-service>
   <link-service-id>mylinkservice</link-service-id>
   <link-service-class>fr.paris.lutece.plugins.mylinkservice.service.MyLinkService</link-service-class>
   <link-service-bean-class>fr.paris.lutece.plugins.mylinkservice.web.MyLinkServiceJspBean</link-service-bean-class>
   <link-service-label>Link to my URIs</link-service-label>
</link-service>
```

DÉRIVATION DE DEFAULTHTMLSERVICE (OU DEFAULTLINKSERVICE)

Dériver **DefaultHTMLService** (resp. DefaultLinkService) en implémentant la méthode **getPluginName** :

```
public class MyHtmlService extends DefaultHtmlService
{
    /** The plugin name. */
    private static final String PLUGIN_NAME = "myhtmlservice";

    public String getPluginName()
    {
         return PLUGIN_NAME;
    }
}
```

IMPLÉMENTATION DE HTMLSERVICESELECTIONBEAN (OU LINKSERVICESELECTIONBEAN)

Il s'agit de fournir un JSPBean qui implémente l'interface HtmlServiceSelectionBean en proposant une méthode getHtmlSelectorUI(), retournant une IHM permettant de sélectionner l'objet à insérer.

exemple de **ImageLibraryJspBean** pour le plugin ImageLibrary.

INSERTION DU CODE HTML

le copier collage du code HTML à insérer se fera en javascript en appelant la méthode **opener.insert_html(strCodeHTMLAInserer)** .

Exemple issu de l'imageLibrary :

```
   ...
   _buffer = _buffer + _hspace + _vspace + _width + _height + _align + ">";

   if (opener != null)
   {
      // The caller must provide an insert_html method
      opener.insert_html(_buffer);
      window.close();
  }
  else
  {
         alert("Editeur HTML indisponible !" );
  }
```

UTILISATION D'UN SERVICE

Un exemple d'utilisation des InsertService est donné par le plugin HTML, par l'intermédiaire du template **editor_portlet_html.html** .

PROPOSER UN LIEN VERS LE SERVICE D’INSERTION

Il s’agit d’appeler la JSP **GetAvailableServices.jsp** , en utilisant par exemple le code suivant :

```
function create_insert_window()
{
  var url="GetAvailableServices.jsp";
  var nom = "Lien ou code HTML";
  child = window.open(url,'','toolbar=no, scrollbars=yes, status=no, location=no, directories=no, menubar=no,
width=450, height=350');
  child.focus();
}

<a href="#" onClick="create insert window();">Insérer Lien ou HTML</a>
```

FOURNIR UNE MÉTHODE JAVASCRIPT INSERT_HTML()

Il s’agit de fournir, dans la page appelant le service d’insertion, une fonction javascript **insert_html** réalisant l’ajout du code HTML retourné par le service d’insertion.

exemple pour le HTML portlet :

```
function insert_html(strHTMLToInsert)
{
  //TEXT
  document.Form.html_content.value = document.Form.html_content.value + strHTMLToInsert;

  //HTML
  theDoc.innerHTML=theDoc.innerHTML+ strHTMLToInsert;
}
```

33. CRÉER UN TRAITEMENT D'ARRIÈRE-PLAN (DAEMON)

Un daemon est un traitement automatique qui s'exécute à intervalles réguliers et de manière asynchrone. Afin de faciliter l'implémentation de ces traitements Lutece dispose d'une interface centralisée de gestion de ces derniers.

Chaque daemon a sa propre fréquence d'exécution et doit fournir un compte-rendu des traitements réalisés qui est consultable dans l'interface de gestion des daemons.

CRÉATION D'UNE CLASSE DAEMON

La classe doit étendre la classe **Daemon** de Lutece et implémenter la méthode **run.**

C'est cette méthode qui sera appelée lors du passage du daemon, à l'intérieur de cette méthode il est nécessaire d'utiliser la méthode **setLastRunLogs** pour fournir le compte rendu d'exécution.

```
import fr.paris.lutece.portal.service.daemon.Daemon;


/**
 * My Daemon
 */
public class MyDaemon extends Daemon
{
 /**
 * {@inheritDoc}
 */
 @Override
 public void run( )
 {
 setLastRunLogs( MyProcess() );
 }
}
```

DÉCLARATION D'UN DAEMON

Un *Daemon* pour qu'il soit chargé par le service de Daemon Lutece doit être déclaré dans le descripteur du plugin.

Cette déclaration se fait en ajoutant la section suivante dans le fichier xml du plugin

```
<!-- Daemons -->

<daemons>
 <daemon>
  <daemon-id>myDaemon</daemon-id>
  <daemon-name>myplugin.daemon.myDaemon.name</daemon-name>
  <daemon-description>myplugin.daemon.myDaemon.description</daemon-description>
  <daemon-class>fr.paris.lutece.plugins.myplugin.service.MyDaemon</daemon-class>
 </daemon>
</daemons>
```

Les propriétés suivantes sont à renseigner :

- daemon-id : La clef technique permettant d'identifier de façon unique un daemon
- daemon-name : La clef i18n (clef contenue dans les fichiers d'internationalisation du plugin) contenant le nom du daeamon
- daemon-description : La clef i18n contenant la description du plugin
- daemon-class : La classe java implémentant le traitement

CONFIGURATION D'UN DAEMON

Les propriétés de configuration d'un daemon doivent être renseignées dans le fichier de configuration du plugin.

Ces propriétés indiquent si le daemon doit etre lancé automatiquement au démarrage de l'application ainsi que son intervalle de passage en seconde.

```
daemon.myDaemon.interval=86400
daemon.myDaemon.onstartup=1
```

Une fois l'application initialisée ces configurations sont stockées dans le Datastore et peuvent être administrées dans le backoffice.

34. CRÉER UNE TÂCHE DE WORKFLOW

Le plugin workflow permet de configurer les traitements que vous souhaitez voir exécuter lorsqu'une action est déclenchée sur une ressource.

Les différents traitements possibles appelés "tâche" sont proposés par les modules de workflow (Notification,génération de pdf...)

Dans le cas ou vous souhaitez créer des tâches spécifiques, il est nécessaire de développer son propre module de workflow.

PRÉ-REQUIS AU DÉVELOPPEMENT D'UN MODULE DE WORKFLOW

Le module de workflow doit dépendre du plugin-workflow, pour ce faire il faut ajouter la section suivante dans le fichier **pom.xml** du module la dépendance suivante:

```
<dependency>
 <groupId>fr.paris.lutece.plugins</groupId>
 <artifactId>plugin-workflow</artifactId>
 <version>[3.0.0,)</version>
 <type>lutece-plugin</type>
</dependency>
```

CRÉATION D'UNE TÂCHE DE WORKFLOW

Pour pouvoir créer une tâche de workflow les classes ci-dessous doivent être développées :

- Une classe **MyTask** étendant la classe **Task** de la librairie library-workflow-core et implémentant principalement la méthode processTask.C'est cette méthode qui sera appelée lors de l'exécution de la tâche
- Une classe **MyTaskConfig** étendant la classe **TaskConfig** de la librairie library-workflow-core et destinéz à contenir la configuration de la tâche
- Une classe **MyTaskConfigDAO** implémentant l'interface **ITaskConfigDAO** de la librairie library-workflow-core et stockant en base de données les propriétés de l'objet **TaskConfig**
- Une classe **MyTaskConfigService** étendant la classe **TaskConfigService** de la librairie library-workflow-core. Cette classe propose des méthodes de création, modification, suppression de configurations.
- Une classe **MyTaskComponent** étendant la classe **TaskComponent** de la librairie library-workflow-core. Cette classe est utilisée principalement en back office pour afficher et enregistrer les informations de configuration et en Front office pour intéragir avec l'utilisateur dans le cas ou la tâche a besoin d'un complément d'information avant de s'exécuter (Affichage d'un formulaire associé à la tâche)

MYTASK

- Créer une classe au niveau de la couche service (fr.paris.lutece.plugins.workflow.modules.mymodule.service.MyTask).
- La nouvelle classe doit étendre la classe abstraite **Task** définie dans la **library-workflow-core** :

```
public class MyTask extends Task
{
 ...
}
```

La library-workflow-core offre une classe abstraite **SimpleTask** héritant de la classe **Task** et implémentant les méthodes de base de cette dernière.

Cette classe peut être utilisée afin de n'avoir à implémenter que les méthodes principales de la classe Task.

MYTASKCONFIG

- Créer une classe au niveau de la couche **business** (fr.paris.lutece.plugins.workflow.modules.mymodule.business.MyTaskConfig).
- La nouvelle classe doit étendre la classe abstraite **TaskConfig** définie dans la library-workflow-core :

```
public class MyTaskConfig extends TaskConfig
{
 ...
}
```

MYTASKCONFIGDAO

- Créer une classe au niveau de la couche **business** (fr.paris.lutece.plugins.workflow.modules.mymodule.business.MyTaskConfigDAO).
- La nouvelle classe doit implémenter l'interface **ITaskConfigDAO** définie dans la library-workflow-core

```
public class MyTaskConfigDAO implements ITaskConfigDAO<MyTaskConfig>
{
 ...
}
```

MYTASKCONFIGSERVICE

- Créer une classe au niveau de la couche service (fr.paris.lutece.plugins.workflow.modules.mymodule.service.MyTaskConfigService).
- La nouvelle classe doit étendre la classe abstraite **TaskConfigService** définie dans la **library-workflow-core** :

```
public class MyTaskConfigService extends TaskConfigService
{
 ...
}
```

MYTASKCOMPONENT

- Créer une classe au niveau de la couche web (fr.paris.lutece.plugins.workflow.modules.mymodule.web.MyTaskTaskComponent). La nouvelle classe doit étendre la classe abstraite TaskComponent définie dans le plugin-workflow :

```
public class MyTaskComponent extends TaskComponent
{
 ...
}
```

- La library-workflow-core offre deux classes abstraites :
  - TaskComponent: utilise BeanUtils pour enregistrer les configurations de façon automatique.
  - SimpleTaskComponent: peut étre utilisée dans le cas ou la tâche développée ne possède ni configuration ni formulaire intermédiaire.
- Le plugin-workflow offre trois autres classes abstraites :
  - AbstractTaskComponent implémente une méthode pour valider les beans utilisant les annotations JSR303.
  - NoFormTaskComponent permettant à votre implémentation d'éviter d'implémenter les méthodes concernant le formulaire intermédiaire.
  - NoConfigTaskComponent permettant à votre implémentation d'éviter d'implémenter les méthodes concernant la configuration de la tâche.

DÉCLARATION DE LA TÂCHE DANS LE CONTEXTE SPRING

Dans le fichier de contexte du module (fichier /WEB-INF/conf/plugins/workflow-mytask_context.xml) il faut déclarer les beans définissant la tâche, son type et sa configuration.

```
<bean id="workflow-mytask.taskTypeMyTask" class="fr.paris.lutece.plugins.workflowcore.business.task.TaskType"
```

```
 p:key="taskMyTask"
 p:titleI18nKey="module.workflow.mytask.task_title"
 p:beanName="workflow-mytask.taskMyTask"
 p:configBeanName="workflow-mytask.taskMyTaskConfig"
 p:configRequired="true"
 p:formTaskRequired="true"
 p:taskForAutomaticAction="true" />

<bean id="workflow-mytask.taskMyTask" class="fr.paris.lutece.plugins.workflow.modules.mytask.service.MyTask"
scope="prototype" />
```

Le bean définissant le type de la tâche doit comporter les attributs suivants

Nom	Description	Exemple
key	La clé technique de la tâche (doit être unique)	taskMyTask
titleI18nKey	La clé i18n contenat le titre de la tâche	module.workflow.mytask.task_title
beanName	Le nom du bean de la tâche (doit être identique à l'ID du bean définissant la tâche)	workflow-mytask.taskMyTask
configBeanName	Le nom du bean de la configuration de la tâche (doit être identique à l'ID du bean définissant la configuration de la tâche). Ne le définir que si la tâche de workflow a besoin d'une configuration	workflow-mytask.taskMyTaskConfig
configRequired	Attribut permettant de savoir si la tâche a besoin d'une configuration de l'administrateur technique ou non. Attribut optionnel si la tâche n'a pas besoin de configuration.	true
formTaskRequired	Attribut permettant de savoir si la tâche a besoin d'un formulaire intermédiaire lors de l'exécution de l'action. Attribut optionnel si la tâche n'a pas besoin de formulaire intermédiaire.	true
taskForAutomaticAction	Attribut permettant de savoir si la tâche peut être assignée à une action automatique. Attribut optionnel si la tâche ne peux pas être assignée à une action automatique.	true

Si la tâche de workflow nécessite une configuration dynamique alors il faut l'ajouter au fichier de contexte

```
<bean id="workflow-mytask.taskMyTaskConfig"
class="fr.paris.lutece.plugins.workflow.modules.mytask.business.MyTaskConfig"
scope="prototype" />

<bean id="workflow-mytask.taskMyTaskConfigService"
class="fr.paris.lutece.plugins.workflow.modules.mytask.service.TaskMyTaskConfigService"
p:taskConfigDAO-ref="workflow-mytask.taskMyTaskConfigDAO" />
```

Si la tâche de workflow nécessite l'affichage d'un formulaire alors il faut l'ajouter au fichier de contexte

```
<bean id="workflow-mytask.myTaskTaskComponent"
class="fr.paris.lutece.plugins.workflow.modules.mytask.web.MyTaskTaskComponent"
 p:taskType-ref="workflow-mytask.taskTypeMyTask"
 p:taskConfigService-ref="workflow-mytask.taskMyTaskConfigService" />
```

DÉVELOPPEMENTS AVANCÉS

35. L'AUTHENTIFICATION FRONT OFFICE (MYLUTECE)
36. CRÉER DES WEB SERVICES REST
37. GÉRER ET DÉVELOPPER DES SERVICES DE CACHES
38. INTÉGRER LE MOTEUR DE WORKFLOW À UN PLUGIN
39. AJOUTER DES EXTENSIONS (COMMENTAIRES, NOTES, VUES...)
40. CONCLUSION

35. L'AUTHENTIFICATION FRONT OFFICE (MYLUTECE)

Plusieurs API ont été mises en place à partir de la version 1.1 de Lutèce pour gérer la sécurité des accès aux ressources du portail. La principale mise en oeuvre est faite par le plugin **MyLutece** qui réalise de manière modulaire l'authentification et la gestion des rôles associés aux utilisateurs authentifiés du site. L'objectif de cette API est de fournir une authentification unique des utilisateurs partagée par tous les plugins d'un site. Ceci signifie que toute application du portail pourra vérifier que l'utilisateur courant est identifié et accéder à ses rôles.

Outre l'authentification des utilisateurs, ces API permettent la réalisation d'applications basées sur des rôles : espaces privés, personnalisation, préférences, rôles applicatifs, ...

API DES MODULES D’AUTHENTIFICATION

La gestion de l’authentification est modulaire et paramétrable, de manière à supporter plusieurs implémentations dont voici une liste non exhaustive :

- Annuaire LDAP (**LDAPAuthentication**)
- Base de données interne (**BaseAuthentication**)
- Authentification déléguée au serveur Web (ex: Tomcat)
- OpenID connect
- Persona (Mozilla)
- FranceConnect (DISIC)
- OpenAM (ForgeRock)
- CAS(Jasig)

L’ensemble des implémentations doit respecter l’interface **fr.paris.lutece.portal.service.security.LutecePortalAuthentication** .

On distingue deux sous-ensembles d’implémentations :

- l’authentification est assurée au niveau de Lutece.
- l’authentification est réalisée en amont de Lutece (ex : serveur web ou WSSO).

Une authentification assurée par Lutece a les caractéristiques suivantes :

- elle doit prévoir des méthodes de connexion/déconnexion (**login** / **logout**).
- elle spécialise la classe abstraite **fr.paris.lutece.plugins.mylutece.authentication.PortalAuthentication** .

Une authentification basée sur un système d'authentification en amont disposera quant à elle des caractéristiques suivantes :

- elle ne doit pas prévoir des méthodes de connexion /déconnexion puisque celles-ci sont assurées par le dispositif externe situé en amont.
- elle spécialise la classe abstraite **fr.paris.lutece.plugins.mylutece.authentication.ExternalAuthentication** .

API DES UTILISATEURS ET GESTION DES COMPTES

A chaque système d’authentification peut être associé un modèle d’utilisateur qui doit implémenter le modèle minimal défini par l’interface **fr.paris.lutece.portal.service.security.LuteceUser** .

Les implémentations notamment prévues sont :

- **LDAPUser**
- **BaseUser**

Ces implémentations ne font pas partie du noyau, elles sont disponibles dans le plugin "mylutece" sous le package **fr.paris.lutece.plugins.mylutece.modules.*** .

La façon dont ces modèles d’utilisateurs délivrent les informations concernant l’utilisateur se rapproche du modèle proposé par la spécification JSR 168, basée elle-même sur celle de la Platform for Privacy Preferences 1.0 réalisée par le W3C. Ainsi les noms et prénoms se récupèrent à partir des clés **user.given.name** et **user.given.family**.

Les mêmes noms d'attributs sont proposés par les OASIS Web Services for Remote Portlets Technical Committee. Voir la liste des attributs.

Concernant la gestion des comptes des utilisateurs, plusieurs modes sont possibles :

- l’utilisateur ne peut pas créer lui-même un compte. La création du compte doit alors être réalisée au niveau de l’annuaire soit par une fonction d’administration intégrée à Lutece soit par un dispositif autre.

- l’utilisateur peut se créer un compte sans nécessiter l’intervention des administrateurs du site. Dans cette situation, l’écran de login affichera un lien vers la page de création d’un nouveau compte.

Dans les deux situations, il sera possible pour l’utilisateur identifié d’accéder à une page optionnelle et paramétrable permettant de visualiser les informations du compte. A partir de cette page et en fonction des choix de gestion des comptes il est envisageable d’ajouter des fonctions de changement de mot de passe ou de modification des informations personnelles.

Concernant les rôles associés aux utilisateurs, ils correspondent à la description faite dans la spécification Java Servlet 2, c’est à dire une notion abstraite définie par une application qui, associée à un utilisateur ou à un groupe d’utilisateur, permet de définir une politique de sécurité.

Remarque importante : selon le système d’authentification utilisé, l’accès à l’ensemble des rôles n’est pas toujours possible. Par exemple, le système d’authentification basé sur le serveur web ne permet que de vérifier qu’un utilisateur dispose bien d’un rôle donné par le biais de la méthode **isUserInRole** , mais la liste des autres rôles n’est pas accessible.

LE SERVICE DE SÉCURITÉ DU NOYAU : SECURITYSERVICE

Ce service est le point d’entrée pour tous les composants : plugins, portlets ou applications désirant obtenir l’identité de l’utilisateur courant.

La principale méthode à utiliser est **getRegisteredUser** qui permet d’obtenir l’identité de l’utilisateur courant enregistré au niveau du portail.

Une nouvelle exception **UserNotSignedException** a été ajoutée à la signature des méthodes **getPage** des interfaces **ContentService** et **XPageApplication** , afin de signifier que l’utilisateur doit être authentifié pour accéder au service ou à l’application. Cette exception est catchée par la page principale du site, **Portal.jsp** . Celle-ci enregistre les paramètres courants de l’url, redirige l’utilisateur vers la page de login paramétrée dans le fichier **mylutece.properties** . Une fois l’identification réalisée, l’utilisateur est redirigé vers l’url à laquelle il tentait d’accéder initialement :

```
LuteceUser user = SecurityService.getInstance().getRegisteredUser( request );

if( user != null )
{
      // The user is authenticated
      ...
}
else
{
     // Throw an exception to force the user to login
     throw new UserNotSignedException();
}
```

A l’instar des interfaces **HttpServletRequest** ou **PortletRequest** (JSR 168), **SecurityService** propose également les méthodes **getRemoteUser** , **getUserPrincipal** et **isUserInRole** .

MISE EN OEUVRE

PARAMÉTRAGE DE L’AUTHENTIFICATION : INSTALLATION DU PLUGIN MYLUTECE

Le plugin doit être déployé comme n’importe quel plugin.

PARAMÉTRAGE DU PLUGIN MYLUTECE

Les paramètres décrivant l’implémentation choisie sont définis dans le fichier de configuration **mylutece.properties** de Lutece grâce aux propriétés suivantes :

```
########################################################
# Authentication management
mylutece.authentication.enable=true
mylutece.authentication.class=fr.paris.lutece.plugins.mylutece.authentication.MultiLuteceAuthentication
```

Description des propriétés :

- **lutece.authentication.enable** : indique si le service d'authentification doit étre activé ou non(obligatoire)
- **lutece.authentication.class** : détermine la classe qui implémente l’authentification choisie (à renseigner si **lutece.authentication.enable** a la valeur **true**).

LES OBJETS DE BASE DU SYSTÈME

L’ÉCRAN D’IDENTIFICATION PAR DÉFAUT

Dans le cas d’une identification interne au portail, le plugin MyLutece propose une page d’identification par défaut :

Cette page est accessible à partir de l’adresse : http://myhost/lutece/jsp/site/Portal.jsp?page=mylutece&action=login.

Son adresse et celle du traitement du formulaire d’identification sont paramétrées dans le fichier **mylutece.properties** comme suit :

```
mylutece.url.login.page=Portal.jsp?page=mylutece&action=login
mylutece.url.doLogin=./plugins/mylutece/DoMyLuteceLogin.jsp
```

LE PORTLET D’IDENTIFICATION

Le plugin MyLutece propose un portlet d’identification.

Lorsque l’utilisateur n’est pas identifié et que le système d’authentification est de type Portail et non externe, le portlet affiche le formulaire de connexion login/password avec des liens facultatifs vers une page de création d’un compte ou de récupération d’un mot de passe oublié.

Une fois l’utilisateur identifié, le portlet affiche un message de bienvenue ainsi que le bouton de déconnexion.

LES PAGES PAR DÉFAUT DE GESTION DU COMPTE

Les pages par défaut concernant la gestion de compte disponibles dans le plugin MyLutece sont les suivantes :

- création d’un nouveau compte
- affichage des informations personnelles
- mot de passe perdu

Les implémentations proposées ne sont pas opérationnelles. Si de telles fonctions de gestion de compte sont à mettre en place, il faudra réaliser une XPageApp ou des JSP pour permettre leur implémentation.

Les fonctions telles que le changement de mot de passe ou la modification des informations personnelles peuvent être implémentées de la même manière et leur accès est à prévoir au niveau de la page d’affichage des informations personnelles.

Le paramétrage des URL de ces fonctions est réalisé dans le fichier **mylutece.properties** . L’exemple suivant montre le paramétrage d’une implémentation réalisée avec une XPageApp nommée **account_manager** :

```
mylutece.url.createAccount.page=Portal.jsp?page=account_manager&action=createAccount
mylutece.url.viewAccount.page=Portal.jsp?page= account_manager&action=viewAccount
mylutece.url.lostPassword.page=Portal.jsp?page= account_manager&action=lostPassword
```

CONTRÔLE DE L’AUTHENTIFICATION POUR LES APPLICATIONS

Il est possible de soumettre l'ensemble des accès au portail à une authentification. Dans ce cas de figure, aucune page ou application n'est accessible à un utilisateur non identifié.

Pour mettre en oeuvre ce mode de fonctionnement il faut activer le filtre d'authentification apporté par le plugin mylutece, en se rendant dans le menu "configuration du filtre d'authentification" dans le back office de Lutece.

APPLICATION DE TEST DE L'AUTHENTIFICATION : PLUGIN MYLUTECETEST

Un plugin **mylutecetest** a été réalisé pour tester et illustrer le comportement d’une application dont l’accès est restreint. Cette application affiche :

- le nom du service d'authentification utilisé.
- les informations disponibles concernant l'utilisateur : clés UserInfo par défaut et clés spécifiques *(voir exemple ci-dessous)*
- les rôles éventuellement possédés par l'utilisateur.

Internal name or login : 9c08b0da44e7747a347ca8f9c3bf71f92370a5a0ccb8d4c1

Email : test@test.com

USER INFO KEY	VALUE
franceconnect.user.accessToken	54442898584c020b95d1aeb65e0f306128ebb14c03dfe75baa255cf220c1f5a7
franceconnect.user.address	26 rue Desaix, 75015 Paris
franceconnect.user.birthCountry	99100
franceconnect.user.birthDate	1976-02-24
franceconnect.user.birthPlace	91272
franceconnect.user.idp.acr	eidas2

franceconnect.user.idp.id	dgfip
user.bdate	1976-02-24
user.business-info.online.email	test@test.com
user.gender	male
user.name.family	Dupond
user.name.given	Pierre
user.name.middle	
user.name.nickName	

User's roles : No role available

This user has been authenticated by : **Lutece FranceConnect Authentication Service**

36. CRÉER DES WEB SERVICES REST

Les services web de type REST (Representational state transfer) exposent entièrement leurs fonctionnalités comme un ensemble de ressources (URI) identifiables et accessibles par la syntaxe et la sémantique du protocole HTTP. Les Services Web de type REST sont donc basés sur l'architecture du web et ses standards de base : HTTP et URI. Les données fournies par ces services sont généralement disponibles en plusieurs formats : XML, JSON ou HTML.

Les opérations de CRUD sur une ressource sont réalisées à l'aide des méthodes HTTP : PUT (create), GET (read), POST (update), DELETE (delete).

Les avantages de ce type de webservices sont :

- Ils sont utilisables par n'importe quelle brique (client riche ou mobile, autre SI, autre service...) dans n'importe quelle techno
- Ils sont exposables via HTTP : API publique et humainement compréhensible
- Ils sont scalables et cachables notamment par les équipements du web : proxys , caches.

CHOIX DE L'IMPLÉMENTATION

CHOIX DU FRAMEWORK

Plusieurs frameworks sont disponibles pour faciliter la réalisation de services web REST. On peut citer notamment Restlet ou Apache CXF. Pour l'implémentation par défaut proposée par Lutece, le framework Jersey a été retenu.

Jersey est l'implémentation de référence de JAX-RS (JSR 311) qui désigne la spécification des web services RESTful.

INTÉGRATION DANS LUTECE : LE PLUGIN REST

Le plugin REST offre une couche de service standard pour tous les modules REST. Il détecte automatiquement toutes les ressources dont la classe est déclarée dans le fichier contexte Spring du plugin ou du module. Cette opération est réalisée au lancement de la webapp et les ressources détectées sont affichées dans les logs.

Le plugin utilise ensuite un filtre de servlet, basé sur l'URI /rest de la webapp et réalise le dispatching vers les URIs des ressources. Lorsque le plugin REST est déployé, il n'y a donc aucun développement ou paramétrage particulier à réaliser en dehors des classes des ressources à fournir et à déclarer dans les fichiers de contexte.

L'utilisation du plugin-rest nécessite donc 3 étapes:

- Ajout de la dépendance vers le composant plugin-rest dans le fichier pom.xml
- Implémentation d'une classe représentant la ressource REST: voir exemple ci-dessous.
- Déclaration de cette classe dans le fichier de contexte de spring (ex webapp/WEB-INF/conf/plugins/<plugin>_context.xml): <bean id=.. class=..>

EXEMPLES DE CODE / ANNOTATIONS

L'écriture de web services est grandement facilitée avec Jersey par le biais d'annotations.

Les principales annotations sont les suivantes : Annotations associées aux méthodes

Type de requête HTTP pris en charge par la méthode	@GET, @PUT, @POST, @DELETE
Chemin de l'URI pris en charge par la méthode	@Path(path)
Format de réponse demandé	@Produces(format) Ex formats : MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML,...
Nature des données en entrée	@Consumes(format) Ex format : MediaType.APPLICATION_FORM_URLENCODED

Voir la documentation de JAX-RS: https://jersey.java.net/documentation/1.18/jax-rs.html

```
@Path(RestConstants.BASE PATH +Constants.PLUGIN NAME )

 public class CustomersResource

{
    @GET
    @Path('customers')
    @Produces( MediaType.APPLICATION_XML )
    public List<Customer> getCustomers()
    {
       ...
    }

    @GET
    @Path('customers/{id}')
    public Customer getCustomer(@PathParam('id') int nId)
    {
       ...
    }

 @PUT
   @Path('customers/add')
  @Produces( MediaType.TEXT_PLAIN )
  @Consumes( MediaType.APPLICATION_XML )
  public String addCustomer(Customer customer)
  {
  ...
  }
}
```

NORMES ET CONVENTIONS DE NOMMAGE

URI

La racine des URI se définit avec l'annotation @Path de la manière suivante :

```
 @Path(RestConstants.BASE_PATH +Constants.PLUGIN_NAME )
```

Ressource :

```
/rest/{plugin}[/{module}]/{ressource}/{id}
```

Liste de ressources :

```
 /rest/{plugin}[/{module}]/{ressource}/
```

Query parameters :

sort, start, count (ou rows)

Java

- Couche REST sous forme de module.
- Nom du module : {plugin}-rest
- Classe correspondant à la ressource. Nom: {Ressource}Rest.java
- Package contenant les classes des ressources. Nom : **.rs**

SÉCURITÉ

La problématique de sécurisation des services web REST réside dans le fait qu'ils sont basés sur un protocole sans état. Chaque requête est indépendante et il n'y a pas de notion de session.

La sécurité, lorsqu'elle doit être mise en oeuvre, doit donc être véhiculée au niveau de chaque requête. Les mécanismes les mieux adaptés sont ceux basés sur des signatures, des tokens ou des clés associés à chaque requête HTTP.

LA LIBRAIRIE SIGNREQUEST

Cette librairie fournit une API pour définir des services d'authentification de requêtes HTTP : les **RequestAuthenticator**. Elle propose plusieurs implémentations dont notamment **HeaderHashRequestAuthenticator** qui permet de créer et valider une signature de la requête basée sur les paramètres de celle-ci et sur un secret partagé entre le client et le serveur. Si la signature est absente, incorrecte ou réalisée avec une mauvaise clé, la requête sera rejetée.

Cette librairie n'a pas de dépendance avec le core de Lutece. Elle peut donc être utilisée facilement par d'autres applications Java non-Lutece comme des applications Android par exemple.

Les implémentations d'authenticator et de filtres fournis par cette librairie peuvent répondre à de nombreuses situations mais sont aussi des exemples. Ils peuvent tout à fait être étendus ou modifiés en fonction des besoins.

LES PARAMÈTRES DE HEADERHASHAUTHENTICATOR

Cet authenticator doit être configuré à l'aide de plusieurs paramètres :

- le service de hachage. La librairie SignRequest fournit une API de HashService et une implémentation utilisant l'algorithme SHA-1.
- la clé privée correspondant au secret partagé entre le client et le serveur
- la liste des paramètres de la requête qui sont utilisés pour composer la signature
- la durée de validité de la signature en secondes. La valeur 0 indique que la durée n'est pas contrôlée.

CONFIGURATION D'UN REQUESTAUTHENTICATOR DANS LE PLUGIN REST

La sécurisation de l'ensemble des requêtes peut se faire au niveau du plugin REST en injectant via le context Spring un authenticator.

Par défaut, le plugin REST utilise l'implémentation NoSecurityRequestAuthenticator qui autorise l'ensemble des requêtes. L'exemple ci-dessous montre une configuration utilisant le HeaderHashRequestAuthenticator et son paramétrage spécifique.

```
<bean id='rest.hashService' class='fr.paris.lutece.util.signrequest.security.Sha1HashService' />
 <bean id='rest.requestAuthenticator' class='fr.paris.lutece.util.signrequest.HeaderHashAuthenticator' >
 <property name='hashService' ref='rest.hashService' />
 <property name='signatureElements' >
 <list>
 <value>key</value>
 </list>
 </property>
 <property name='privateKey'>
 <value>change me</value>
 </property>
 <property name='validityTimePeriod'>
 <value>0</value>
 </property>
 </bean>
```

Un autre RequestAuthenticator dénommé RequestHashAthenticator est disponible dans la librairie. La seule différence est que la signature et le timestamp sont passés en paramètre de la requête HTTP au lieu de Headers de celle-ci. La signature est moins masquée, mais cela permet de sécuriser des requêtes réalisées par des liens hypertexte.

CONFIGURATION D'UN FILTRE DE SERVLET LUTECE

Dans le fichier XML d'un module REST il est possible de déclarer un filtre qui fournira une sécurité spécifique aux ressources du module de la manière suivante :

```
<filters>
 <filter>
 <filter-name>myresourcesecurity</filter-name>
 <url-pattern>/rest/myresource/*</url-pattern>
 <filter-class>fr.paris.lutece.util.signrequest.servlet.HeaderHashRequestFilter</filter-class>
 <init-param>
 <param-name>elementsSignature</param-name>
 <param-value>id-resource,name,description</param-value>
 </init-param>
 <init-param>
 <param-name>validityTimePeriod</param-name>
 <param-value>0</param-value>
 </init-param>
 <init-param>
 <param-name>privateKey</param-name>
 <param-value>change me</param-value>
 </init-param>
 </filter>
</filters>
```

37. GÉRER ET DÉVELOPPER DES SERVICES DE CACHES

La gestion du cache est un élément essentiel pour garantir des performances optimales de Lutece.

Nombre d'opérations de construction des pages sont coûteuses en CPU. On peut citer notamment toutes les opérations mettant en oeuvre de la transformation XML/XSL ou du parsing ou des accès à la base données. Sans cache, Lutece ne pourrait supporter que quelques utilisateurs avec des temps de réponse très médiocres. Faire appel au cache est donc absolument nécessaire !

Les données stockées dans le cache occupent de l'espace mémoire donc leur volume est un paramètre a garder toujours à l'esprit.

Par ailleurs, les données du cache ne sont qu'un reflet des données réelles. Dans certains cas on tiendra à ce qu'elles soient toujours en phase avec les données réelles : le cache devra être "invalidé" à chaque modification d'une donnée ou celles-ci ne sont pas "cachées". Dans d'autres cas, c'est la durée de vie des objets en cache qui donnera la limite acceptable pour le délai de mise à jour des données.

Ce sont toutes ces considérations qu'il s'agit de discuter dans cet article pour configurer Lutece dans une optique de performances maximales.

PRINCIPES DE BASE DES CACHES

L'algorithme de base d'une gestion de cache est relativement trivial. Le cache est un grand réservoir en mémoire dans lequel on stocke des objets dont chacun est identifié par une clé unique calculée. Le modèle de collection utilisé en général pour stocker un cache est une Map.

L'algorithme de récupération d'un objet est donc le suivant :

- on calcule sa clé de cache (hash, identifiant, ou combinaison de clés)
- on recherche l'objet dans le cache
- si l'objet est trouvé
  - on le renvoie
- sinon
  - on récupère ou on recalcule l'objet réel,
  - on met l'objet dans le cache associé à sa clé
  - on renvoie l'objet

La construction de la clé de cache est donc un aspect important de la gestion.

Par ailleurs, les services de Lutece s'appuient sur le produit Ehcache qui permet une gestion plus fine des caches en introduisant des notions telles que :

nombre maximum d'objets, durée de vie des objets, stockage sur disque ...

UNE INTERFACE CENTRALISÉE DE GESTION DES CACHES

Lutece offre une interface centralisée de gestion de tous les services de cache, accessible à l'administrateur du site.

Cette interface liste les services de cache installés, leur statut, les principales options de configuration et des éléments dynamiques d'utilisation (nombre d'objets, mémoire occupée, ...).

LES DIFFÉRENTS NIVEAUX DE CACHE

Les dispositifs de cache sont présents à plusieurs niveaux et ils peuvent apporter des gains pour accéder à des ressources lourdes à calculer ou à charger.

On distingue deux principaux niveaux.

LES SERVICES DE CACHE DE PREMIER NIVEAU

Ces services sont implémentés sous forme de filtre de servlet. C'est à dire qu'ils interceptent tout appel HTTP à une ressource (jsp, js, images, ...).

Ces services sont basés sur la brique Ehcache-Web. Les clés de cache sont construites à partir des éléments de la requête HTTP : method, uri path, querystring.

Ces services fournissent d'autres optimisations :

- compression GZIP des réponses
- headers HTTP adaptés à la durée de vie des objets pour profiter au mieux des caches des navigateurs et des proxys notamment pour les ressources statiques (images, css, scripts, ...).

LES SERVICES DE CACHE DE SECOND NIVEAU (CACHE DES RESSOURCES)

Ces services assurent le cache des différents objets Lutece : page, portlet, menus, arborescence du site, document.

CONFIGURATION DES CACHES

Deux fichiers situés dans le répertoire WEB-INF/conf/ permettent de gérer la configuration des caches :

- caches.properties
- caches.dat

caches.properties

Ce fichier contient le paramétrage par défaut des caches.

```
# Default cache configuration
 lutece.cache.default.maxElementsInMemory=10000
 lutece.cache.default.eternal=flase
 lutece.cache.default.timeToIdleSeconds=10000
 lutece.cache.default.timeToLiveSeconds=10000
 lutece.cache.default.overflowToDisk=true
 lutece.cache.default.diskPersistent=true
 lutece.cache.default.diskExpiryThreadIntervalSeconds=120
 lutece.cache.default.maxElementsOnDisk=10000
```

caches.dat

Ce fichier contient le statut et le paramétrage spécifique des caches au démarrage de la Webapp.

```
#Caches status file
 #Sun Mar 27 03:06:48 CEST 2011
 SiteMapService.enabled=1
 MyPortalWidgetService.enabled=1
 PortalMenuService.enabled=1
 DocumentResourceServletCache.enabled=1
 PageCacheService.enabled=1
 PortletCacheService.enabled=1
 MyPortalWidgetContentService.enabled=1
 PageCachingFilter.enabled=1
 StaticFilesCachingFilter.enabled=1
 StaticFilesCachingFilter.timeToLiveSeconds=1000000
```

La racine des propriétés est le nom des caches (avec les espaces supprimés si le nom original en contient). La propriété enabled indique si le cache est activé (valeur=1) ou désactivé (valeur=0). Si cette propriété est absente, le cache est activé par défaut.

Les autres propriétés sont les mêmes paramètres de cache que ceux du fichier caches.properties. La valeur indiquée dans ce fichier surchargera la valeur par défaut définie dans caches.properties.

A partir de la version 4.1 ces infos sont sauvegardées en base dans le Datastore. Les fichiers caches.properties et caches.dat ne servent que pour l'initialisation des valeurs.

STRATÉGIES DE CONFIGURATION

Comme indiqué en introduction, il n'y a pas de configuration universelle. Il s'agit de trouver les meilleurs compromis entre fraicheur des données, performances et consommation de ressources mémoire.

Voici les principales configurations.

CONFIGURATION DE DÉVELOPPEMENT

En développement, il n'y a pas d'enjeux de performances et il est préférable de suivre la mise à jour des données sans effet de cache. La configuration à retenir est donc de désactiver tous les caches.

```
 StaticFilesCachingFilter.enabled=0
 PageCachingFilter.enabled=0
 PageCacheService.enabled=0
 PortletCacheService.enabled=0
 PortalMenuService.enabled=0
 SiteMapService.enabled=0
```

il est possible également de désactiver l'ensemble des caches en modifiant une propriété system au niveau du lancement de

la VM Java.

```
java -Dnet.sf.ehcache.disabled=true
```

CONFIGURATION DE PRODUCTION POUR SITE SANS PERSONNALISATION

Le cache de premier niveau peut être activé à la fois sur les ressources statiques (images, css, scripts) mais aussi sur les pages du portail dont notamment Portal.jsp.

La durée de vie du cache des ressources statiques peut être configurée à une semaine soit 604800 secondes. Celle des pages JSP peut être configurée à une heure soit 3600 secondes

```
StaticFilesCachingFilter.enabled=1
StaticFilesCachingFilter.timeToLiveSeconds=604800
PageCachingFilter.enabled=1
PageCachingFilter.timeToLiveSeconds=3600
```

Les caches de deuxième niveau (pages, portlets, menu,...) ne sont pas obligatoires dans ce cas. Si ils sont activés, la durée de vie des objets doit être cohérente avec celle du cache de premier niveau, soit dans le cas présent la valeur à retenir serait également une heure.

```
PageCacheService.enabled=0
PortletCacheService.enabled=0
PortalMenuService.enabled=0
SiteMapService.enabled=0
DocumentResourceServletCache.enabled=1
```

CONFIGURATION DE PRODUCTION POUR SITE AVEC PERSONNALISATION

La personnalisation empêche d'utiliser le cache de premier niveau pour les pages JSP. En effet la même page ne s'affichant pas de la même manière en fonction de l'utilisateur connecté, il n'est pas possible de servir une page à partir de sa seule adresse.

Pour assurer un cache efficace des pages, il est alors nécessaire de s'appuyer sur les caches de second niveau qui peuvent gérer dans leurs clés l'identifiant de l'utilisateur.

```
StaticFilesCachingFilter.enabled=1
 StaticFilesCachingFilter.timeToLiveSeconds=604800
 PageCachingFilter.enabled=0
 PageCacheService.enabled=1
 PortletCacheService.enabled=1
 PortalMenuService.enabled=1
 SiteMapService.enabled=1
 DocumentResourceServletCache.enabled=1
```

Dans ce type de configuration, les identifiants des utilisateurs étant dans la clé de cache, il faut impérativement tenir compte du nombre d'utilisateurs pour tous les aspects de dimensionnement. Il peut être envisagé de créer des clés de cache spécifiques en fonction du mode de personnalisation pour réduire le nombre d'objets et les performances.

CONFIGURATION DE PRODUCTION POUR ENVIRONNEMENT LIMITÉ EN MÉMOIRE

Dans un contexte où la mémoire disponible est faible et/ou la taille des ressources est importante, il convient de dimensionner correctement le paramètre maxElementsInMemory. Si le nombre d'objets en cache dépasse cette valeur, les objets les moins récemment utilisés (eviction policy LRU Least Recently Used) seront déplacés vers un cache disque, limitant ainsi les risques de dépassement mémoire.

DÉVELOPPEMENT D'UN SERVICE DE CACHE

Pour développer un service de cache intégré à Lutece il suffit d'étendre la classe abstraite AbstractCacheableService.

```
public class MyCacheService extends AbstractCacheableService
{
  private static final String SERVICE_NAME = "My Cache Service";

  public MyCacheService()
  {
  initCache();
  }

 public String getName( )
  {
  return SERVICE_NAME;
  }

 public MyResource getResource( String strId )
  {
  MyResource r = getFromCache( strId );
  if( r == null )
  {
  r = getResourceFromSource( strId );
  putInCache( strId , r );
  }
   return r;
  }
}
```

CONSTRUCTION DES CLÉS DE CACHE

Pour rendre les clés faciles à lire, la norme retenue est de concaténer les éléments sous la forme [key1:value1][key2:value2]...[user:id].

Par ailleurs, ces constructions étant présumées être fortement sollicitées, on utilisera une concaténation avec un `StringBuilder`.

Voici une implémentation typique respectant ces normes :

```
private String getCacheKey( String strId , LuteceUser user )
{
 StringBuilder sbKey = new StringBuilder();
 sbKey.append( "[res:" ).append( strId ).append( "][user:").append( user.getName() ).append("]");
 return sbKey.toString();
}
```

L'INTERFACE ICACHEKEYSERVICE ET L'IMPLEMENTATION DEFAULTCACHEKEYSERVICE

Certains services tels que `PageService` acceptent l'injection par le biais du contexte Spring d'une classe implémentant l'interface `ICacheKeyService` permettant de générer des clés de cache à partir d'une map de paramètres et de l'objet user.

La classe `DefaultCacheKeyService` propose une implémentation par défaut de cette interface. L'interface prévoit de définir pour la génération de la clé :

- une liste de paramètres à utiliser pour générer la clé. Il s'agit d'une mesure de sécurité pour empêcher qu'un générateur utilisant des paramètres fictifs dans les urls ne génère autant de clés de cache qui viendrait à saturer la mémoire.

- une liste de paramètres à ignorer. Certains paramètres peuvent ne pas être pertinents dans la construction et peuvent générer des doublons dans le cache. Il convient donc de les éliminer en les déclarant dans cette liste.

Voici l'exemple d'injection de `CacheKeyService` dans un contexte Spring pour `PageService` :

```
<bean id="pageCacheKeyService" class="fr.paris.lutece.portal.service.cache.DefaultCacheKeyService" >
 <property name="allowedParametersList" >
```

```
  <list>
  <value>page_id</value>
  </list>
  </property>
</bean>

<bean id="portletCacheKeyService" class="fr.paris.lutece.portal.service.cache.DefaultCacheKeyService" >
  <property name="ignoredParametersList" >
  <list>
  <value>page-id</value>
  <value>site-path</value>
  </list>
  </property>
 </bean>

<bean id="pageService" >
  <property name="pageCacheKeyService" ref="pageCacheKeyService" />
  <property name="portletCacheKeyService" ref="portletCacheKeyService" />
</bean>
```

38. INTÉGRER LE MOTEUR DE WORKFLOW À UN PLUGIN

UTILISER LES SERVICES DE WORKFLOW DANS UN PLUGIN

Le service de workflow de lutece permet de gèrer le cycle de vie des ressources d'un site. il est notamment utilisé par des plugins destinés à gérer des demandes utilisateurs tels que directory, rendez-vous ...

Pour utiliser ce service dans un plugin, il est nécessaire de fournir au moteur de workflow un type de ressource sur lequel sera appliqué le ou les workflows ainsi qu'un identifiant de ressource.

Dans l'exemple du plugin directory, afin de connaitre l'état d'une demande, le plugin fournit au moteur de workflow l'identifiant de la demande ainsi que la chaine de caractères "DIRECTORY_RECORD" représentant les fiches d'un directory.

INITIALISER/OBTENIR L'ÉTAT D'UNE RESSOURCE

Afin d'obtenir l'état d'une ressource il faut faire appel à la méthode getState du service de workflow en fournissant l'identifiant de la ressource, le type de ressource, l'identifiant du workflow.

Dans le cas ou la ressource n'a pas d'état, le service de workflow lui affecte l'état initial. Les différents états d'un workflow ainsi que les transitions sont administrables dans les interfaces back office du plugin workflow.

```
import fr.paris.lutece.portal.service.workflow.WorkflowService;

WorkflowService.getInstance( ).getState( idResource, WORKFLOW RESOURCE TYPE, idWorkflow, -1 );
```

OBTENIR LES ACTIONS DISPONIBLES POUR UNE RESSOURCE

Afin d'obtenir l'ensemble des actions disponibles pour une ressource en fonction de son état, il faut faire appel à la méthode getActions du service de workflow en fournissant l'identifiant de la ressource, le type de ressource, l'identifiant du workflow ainsi que l'objet Lutèce représentant l'utilisateur souhaitant récupérer ces informations.

```
import fr.paris.lutece.portal.service.workflow.WorkflowService;

WorkflowService.getInstance( ).getActions( idResource, RESOURCE TYPE, idWorkflow, getUser( ) );
```

EXÉCUTER UNE ACTION SUR UNE RESSOURCE

Pour déclencher une action sur une ressource il est nécessaire de faire appel à la méthode doProcessAction en fournissant l'identifiant de ressource, le type de ressource, l'identifiant de l'action ainsi qu'un paramètre indiquant s'il est nécessaire que le service de workflow vérifie que l'utilisateur est autorisé à effectuer cette action (dans l'appel ci dessous c'est la variable isAutomatic qui fournit cette information).

```
import fr.paris.lutece.portal.service.workflow.WorkflowService;


WorkflowService.getInstance().doProcessAction( idResource, WORKFLOW_RESOURCE_TYPE, idAction,-1, request, locale,
isAutomatic );
```

DÉCLENCHEMENT DES ACTIONS AUTOMATIQUES ASSOCIÉES À UNE RESSOURCE

Le plugin workflow permet d'associer à un état des actions automatiques. Lorsqu'une ressource arrive dans un état possédant des actions automatiques, ces dernières sont éxécutées. Ces traitements sont effectués de manière asynchrone par un daemon du plugin workflow.

Dans le cas où il est nécessaire que ces actions soient déclenchées de manière immédiate, la méthode executeActionAutomatic doit être utilisée.

Cette méthode prend en paramètre l'identifiant de la ressource, le type de ressource et l'identifiant du workflow.

```
import fr.paris.lutece.portal.service.workflow.WorkflowService;

WorkflowService.getInstance( ).executeActionAutomatic( idResource, WORKFLOW RESOURCE TYPE, idWorkflow, -1 )
```

RÉCUPÉRER L'HISTORIQUE D'UNE RESSOURCE

Il est possible de récupérer l'historique des actions et des états d'une ressource en utilisant la méthode `getDisplayDocumentHistory`.

```
import fr.paris.lutece.portal.service.workflow.WorkflowService;


WorkflowService.getInstance( ).getDisplayDocumentHistory( idResource, WORKFLOW_RESOURCE_TYPE, idWorkflow, request,
getLocale( ) );
```

GÉRER LES WORKFLOWS ASSOCIÉS À UN PLUGIN

Pour pouvoir créer des workflows il faut d'abord s'assurer que le plugin-workflow est installé et activé dans le back office de Lutèce.

Pour cela il faut avoir ajouté la dépendance suivante dans le fichier pom.xml de son site.

```
 <dependency>
  <groupId>fr.paris.lutece.plugins</groupId>
  <artifactId>plugin-workflow</artifactId>
  <version>[3.0.0,)</version>
  <type>lutece-plugin</type>
 </dependency>
```

39. AJOUTER DES EXTENSIONS (COMMENTAIRES, NOTES, VUES...)

INTRODUCTION

Le plugin-extend permet aux internautes d'interagir avec des ressources Lutèce, comme par exemple des documents. Il permet en outre de :

- voir le nombre de vues sur une ressource ;
- commenter une ressource ;
- envoyer une notification aux webmestres pour indiquer leur avis sur la ressource (prochainement) ;
- interagir avec des réseaux sociaux (prochainement).

INTÉGRATION

Afin de faciliter la compréhension du guide, l'intégration de extend se fera à partir d'un exemple : intégration de extend sur le plugin-document.

IMPLÉMENTATION DE L'INTERFACE IEXTENDABLERESOURCE

Faire implémenter l'interface IExtendableResource à la classe représentative de la ressource Document. La classe en question, Document, doit alors implémenter les méthodes getIdExtendableResource, getExtendableResourceType et getExtendableResourceName :

```
public class Document implements Localizable, IExtendableResource
  {
  ...
        /**
     * {@inheritDoc}
     */
    @Override
    public String getIdExtendableResource( )
    {
        return Integer.toString( _nIdDocument );
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public String getExtendableResourceType( )
    {
        return PROPERTY_RESOURCE_TYPE;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public String getExtendableResourceName( )
    {
        return _strTitle;
    }
        ...
  }
```

IMPLÉMENTATION DU SERVICE DOCUMENTEXTENDABLERESOURCESERVICE

Ce service permet de récupérer la ressource à partir d'un type de ressource et de son ID. Il permet également de déclarer le type de ressource "Document" auprès du plugin extend :

```
public class DocumentExtendableResourceService implements IExtendableResourceService
 {
    /**
     * {@inheritDoc}
     */
    @Override
    public boolean isInvoked( String strResourceType )
    {
        return Document.PROPERTY_RESOURCE_TYPE.equals( strResourceType );
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public IExtendableResource getResource( String strIdResource, String strResourceType )
    {
        if ( StringUtils.isNotBlank( strIdResource ) && StringUtils.isNumeric( strIdResource ) )
        {
            int nIdDocument = Integer.parseInt( strIdResource );
            return DocumentHome.findByPrimaryKey( nIdDocument );
        }
        return null;
    }

        /**
     * {@inheritDoc}
     */
    @Override
    public String getResourceType( )
    {
        return Document.PROPERTY_RESOURCE_TYPE;
    }

        /**
     * {@inheritDoc}
     */
    @Override
    public String getResourceTypeDescription( Locale locale )
    {
        return I18nService.getLocalizedString( MESSAGE_DOCUMENT_RESOURCE_TYPE_DESCRIPTION, locale );
    }
}
```

Ce service est instancié par Spring. Il faut alors le déclarer au niveau du fichier document_context.xml :

```
<bean id="document.documentExtendableResourceService"
class="fr.paris.lutece.plugins.document.service.resource.DocumentExtendableResourceService" />
```

Attention, si le plugin est désactivé, alors la ressource ne sera pas disponible auprès du plugin extend !

AJOUT DES BOUTONS EN BO POUR INTERAGIR AVEC EXTEND

Modification de la JSPBean

Dans la méthode getModifyDocument, y ajouter la ligne suivante :

```
public String getModifyDocument( HttpServletRequest request )
  {
       ...
       ExtendableResourcePluginActionManager.fillModel( request, getUser( ), model, strDocumentId,
Document.PROPERTY_RESOURCE_TYPE );
       ...
  }
```

Cette appel va ajouter dans le modèle les boutons qui vont bien.

Modification du template

Modifier le template modify_document.html et y ajouter la ligne suivante :

```
${extendableResourceActionsHtml!}
```

AFFICHAGE DES CONTENUS EN FO

Pour afficher un contenu particulier en front office, il faut ajouter le marker suivant dans un template :

```
@Extenter[<idExtendableResource>,<resourceType>,<extenderType>,<parameters>]@
```

où :

```
<idExtendableResource> correspond à l'ID de la ressource (ex : 1)
    <resourceType> correspond au type de ressource. Exemples (attention, le type est sensible à la casse) :
```

Type de ressource	Description
PAGE	Il s'agit des pages classiques d'un portail Lutèce
document	Il s'agit des documents du plugin-document

<extenderType> correspond au type de l'extension. Exemples (attention, le type est sensible à la casse) :

Type d'extension	Description
comment	Pour commenter les ressources
hit	Pour le nombre de vues sur la ressource
rating	Pour voter pour les ressources
feedback	Pour donner son avis sur les ressources

<parameters> correspond aux paramètres supplémentaires à ajouter pour les extensions (ex : {show:true})

Quelques exemples :

Markeur	Description
@ Extender[1,document,comment]@	Affiche les commentaires pour le document ayant l'ID 1.
@ Extender[1,document,hit,{show:true}]@]@	Affiche le nombre de vues sur le document ayant l'ID 1.
@ Extender[1,document,hit,{show:false}]@	N'affiche pas le nombre de vues pour le document ayant l'ID 1, mais incrémente le compteur.
@ Extender[1,document,hit,{increment:true}]@	incrémente le compteur
@ Extender[1,document,hit,{increment:false}]@	n'incrémente pas le compteur
@ Extender[1,document,rating,{show:"vote"}]@	Affiche la note du document ayant l'ID 1@
@ Extender[1,document,rating,{show:"voteAction"}]@	Affiche les liens pour voter le document ayant l'ID 1
@ Extender[1,document,rating,{show:"all"}]@	Affiche la note et les liens du document ayant l'ID 1
@ Extender[2,PAGE,feedback]@	Affiche un formulaire pour donner son avis sur la page ayant l'ID 2

MARKEURS AVEC ID PARAMÉTRABLES

Dans le cas où le traitement des extentions est fait côté serveur, il est possible d'utiliser des ID de ressources paramétrables. Dans ce cas, l'ID doit toujours être défini dans un template HTML, mais peut être défini à n'importe quel endroit. Cela permet par exemple de mettre un markeur dans le header (qui est géré par le coeur Lutèce), et de ne définir l'ID de la ressource que dans le body (par exemple dans un template géré par un plugin).

Pour faire cela, il suffit, au lieu de mettre l'identifiant, de mettre ExtendParameteredId.

L'id doit ensuite être défini dans un markeur :

```
@ ExtenterParameter[<idExtendableResource>,<resourceType>,<extenderType>]@
```

Si l'ID n'est pas spécifié par un markeur ExtenderParameter, alors le markeur d'origine est ignoré.

Exemple :

Afin d'ajouter dans l'en-tête les meta données liées à Open graph, on ajoute dans la balise <head> du fichier page_framset.html le markeur :

```
@ Extender[ExtendParameteredId,document,opengraph,{header:true}]@
```

On ajouter ensuite par exemple dans le template page_template_document1.html du plugin Document le markeur de paramètre suivant :

```
@ ExtenderParameter[1,document,opengraph]@
```

Ainsi, si le contenu du template page_template_document1.html est inclus dans la page finale, alors le premier markeur sera remplacé par les meta données relatives au document d'id 1, et seront placées dans la balise <head>.

SUPPRESSION DE RESSOURCE

Lorsqu'une ressource extensible (dans notre exemple, un document) est supprimée, il faut notifier les différentes extensions de la suppression de cette ressource. Pour cela, il suffit d'appeler la méthode fr.paris.lutece.portal.service.resource.ExtendableResourceRemovalListenerService.doRemoveResourceExtentions(String strExtendableResourceType, String strExtendableResourceId). Cette méthode se chargera de supprimer toutes les données contenues en base relative à la ressource supprimée.

Le premier paramètre de cette méthode est le type de la ressource supprimée (par exemple "document" dans notre cas). Le deuxième paramètre est l'identifiant de la ressource.

INTÉGRER LE CONTENU DE L'ÉCRAN D'INFORMATION D'UNE EXTENTION DANS UNE XPAGE

Certains des modules du plugin Extend permettent de visualiser des informations relatives à l'extension (ex : le module extend comment permet de visualiser les commentaires d'une ressource dans la page de gestion des informations de l'extension). Cette page d'information de l'extension peut être intégrée dans une XPage d'un plugin. Pour cela, il suffit de récupérer le contenu de la page d'information grace au ResourceExtenderComponent de ce module.

Par exemple, la méthode suivante permet d'afficher de façon autonome la page d'information du module extend comment :

```
public String getPage( HttpServletRequest request )

{

        ResourceExtenderDTO resourceExtender = new ResourceExtenderDTO( );
        String strIdExtendableResource = request.getParameter( MARK_ID_EXTENDABLE_RESOURCE ); // Votre marqueur
permettant d'obtenir l'identifiant de la ressource
        String strExtendableResourceType = request.getParameter( MARK_EXTENDABLE_RESOURCE_TYPE ); // Votre marqueur
permettant d'obtenir le type de la ressource
        resourceExtender.setIdExtendableResource( strIdExtendableResource );
        resourceExtender.setExtendableResourceType( strExtendableResourceType );
        resourceExtender.setExtenderType( EXTENDER_TYPE_COMMENT ); // Le type d'extension que vous souhaitez afficher
(ici comment)

        // On récupère ensuite le ResourceExtenderComponent du module désiré
        IResourceExtenderComponentManager extenderComponentManager = SpringContextService.getBean(
ResourceExtenderComponentManager.BEAN_MANAGER );
        IResourceExtenderComponent resourceExtendercomponent = extenderComponentManager.getResourceExtenderComponent(
EXTENDER_TYPE_COMMENT );

        // On récupère le contenu HTML de la page d'information
        String strHtml = resourceExtendercomponent.getInfoHtml( resourceExtender, AdminUserService.getLocale( request
), request );

        return getAdminPage( strHtml );

}
```

40. CONCLUSION

Ce livre est loin d'être exhaustif. Il avait pour principal but de donner les clés pour démarrage en douceur. Notre objectif était de montrer le potentiel de modularité et d'ouverture de Lutece. Vous pourrez ainsi réaliser des portails ou des applications sur mesure et très riches tout en gardant une maîtrise optimale du code. Nous espérons que ce livre vous aura donné des idées pour créer de nouveaux services adaptés précisement à vos besoins.