ACube

Guide du développeur

Version 1.7 du 22/04/2010

Etat : en cours

SUIVI DES MODIFICATIONS

Version	Rédaction	Description	Vérification	Date
0.1	D. WARNIER	Initialisation à partir du guide ancien format		
0.2	G. PICAVET	Partie serveur		
0.3	G.PICAVET	Retours développeurs eds		
0.4	G.PICAVET	Tests unitaires, nomenclature, performances et bonnes pratiques		10/07/08
0.5	G.PICAVET	Gestion de l'upload de fichiers		15/07/08
0.6	M. Fraudeau	Contenttype xsl et struts2		03/09/08
0.7	G.PICAVET	Réécriture paragraphe « Upload/Download de fichiers » + « gestion BLOB et CLOB» + « gestion de la Session »		29/09/08
0.8	C. Rocheteau A. Lesuffleur+ G.PICAVET + C. Richomme	Pagination + limiter le nombre de résultat SQL + Les alias de struts		14/10/08
0.9	K. COIFFET + A. Lesuffleur	Internationalisation avec Spring Modification des datasources + Correction de la déclaration de maMethode dans struts.xml		
1.0	R.QUERE	Correction initialisation des services dans les actions + principes à éviter dans l'utilisation des services		09/01/09
1.1	K.COIFFET	Détail concernant l'appel aux procédures stockées avec Ibatis		19/03/09
1.2	K. COIFFET	Type Java spécifique pour la partie Ibatis et Struts2		12/05/09
1.3	K.LEBLANC	JasperReports + IReport		24/07/09
1.4	A. Lesuffleur	Ajout spécifité LISE 3.1 (relecture, gestion du cache)		06/11/09
1.5	A. Lesuffleur, R Quere	Prise en compte des remarques du MAEE, suppression des TODO		04/12/09
1.6	G. PICAVET C. Richomme A. Lesuffleur	LISE 3.2 : FDF §3.8.2 Correction config. Autorisation §3.3.6.4 Ajout utilisation du captcha §3.9 Ajout des Template de Mail §3.8.5		13/01/10 16/04/10
1.7	A. Lesuffleur	Correction suite aux remarques MAEE du 22/04/10		22/04/10
		Document validé dans sa version xxx		

LISTE DE DIFFUSION

Organisation	Nom	Info	Commentaire	Validation

SOMMAIRE

TABLEAUX

Aucune entrée de table d'illustration n'a été trouvée.

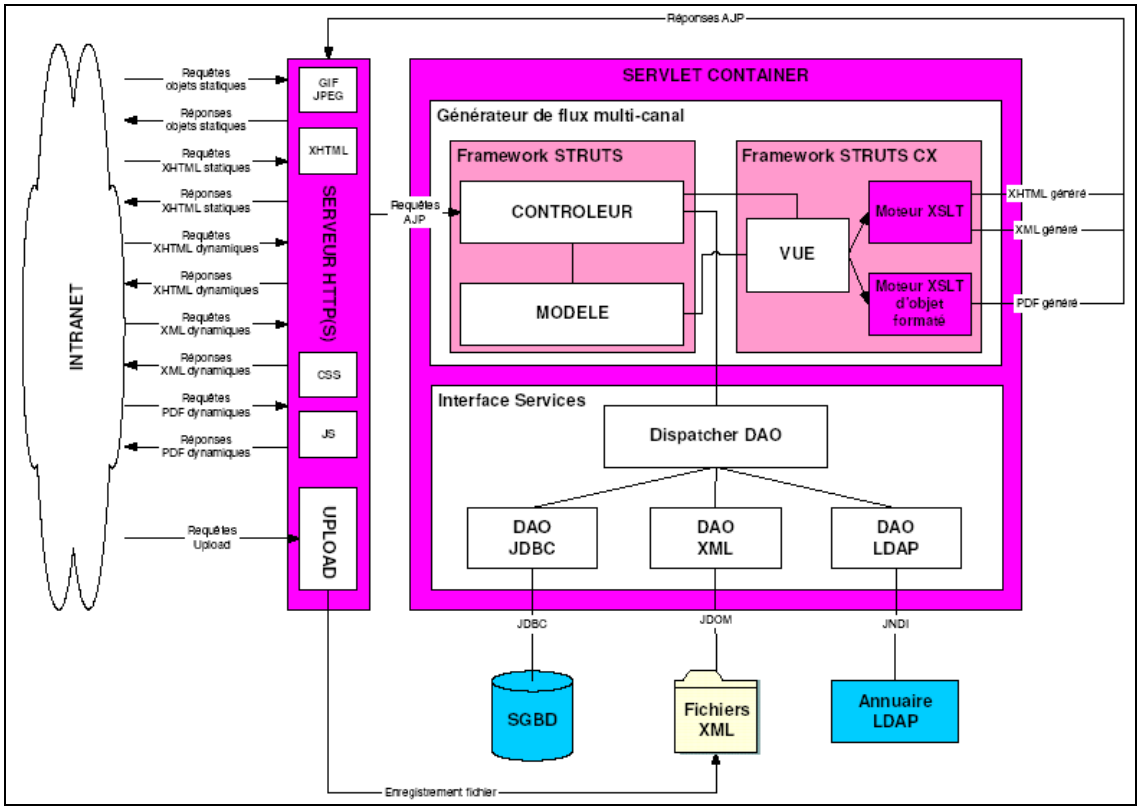
FIGURES

DOCUMENTS DE REFERENCE

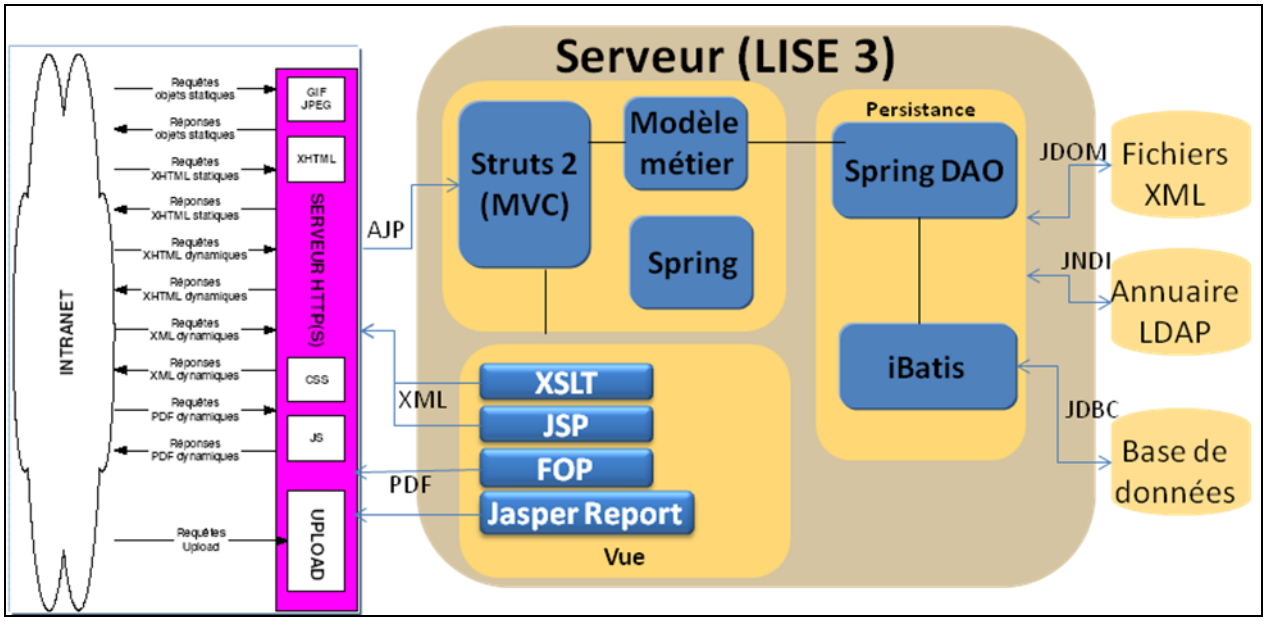
Version	Titre

1 PRESENTATION DE L'ARCHITECTURE A3

1.1 RAPPEL DE L'ARCHITECTURE ANTERIEURE

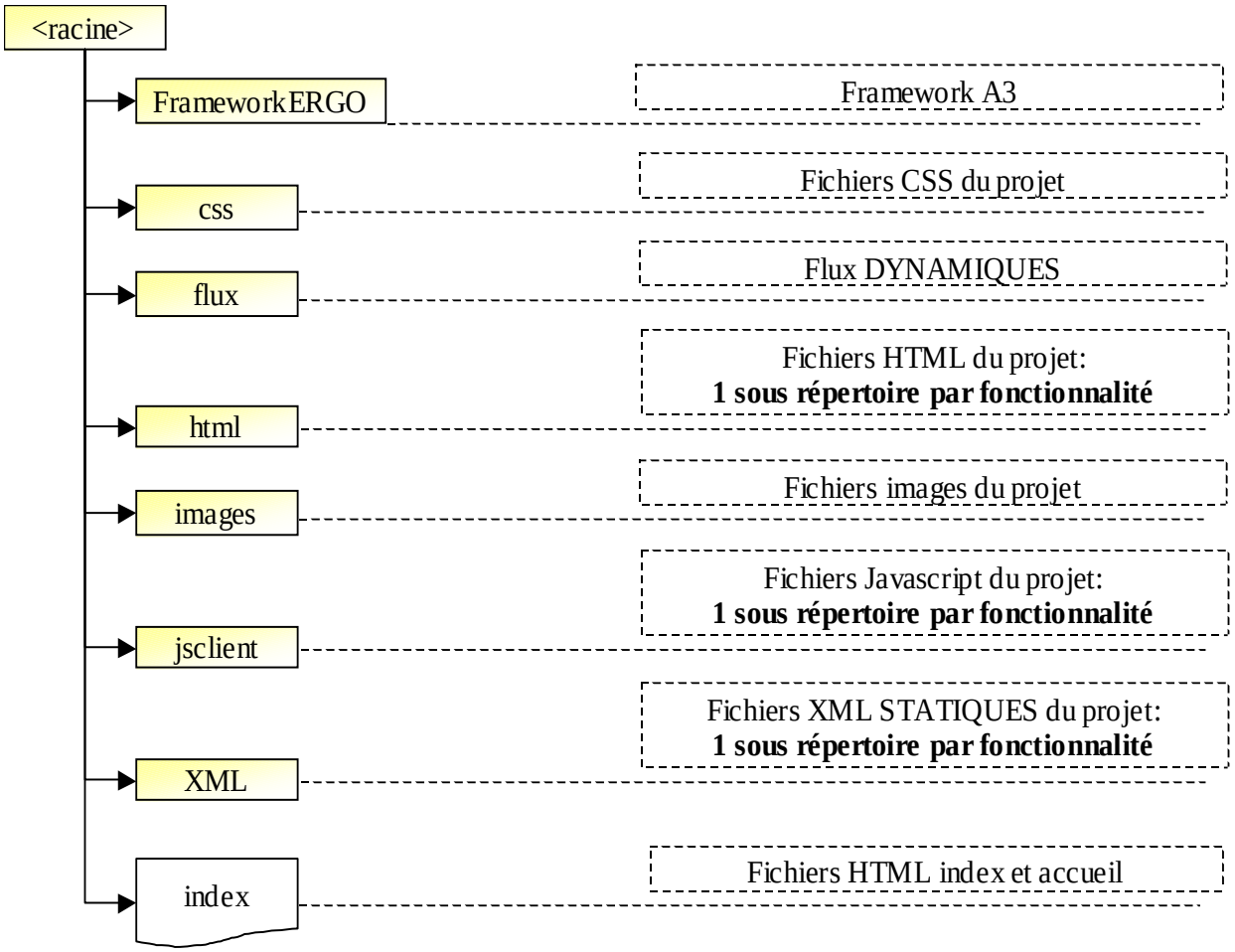


1.2 NOUVELLE ARCHITECTURE



2 DEVELOPPEMENT CLIENT

2.1 ORGANISATION DU CODE



2.2 FLUX XML STATIQUES

Les flux XML « statiques » (libellés des pages html, structures des tableaux, etc …) nécessaires à la construction des pages html sont contenus dans des fichiers XML sous le répertoire <racine>/XML/.
Il est recommander de mutualiser les libellés d'une même sous fonctionnalité dans un seul fichier XML.

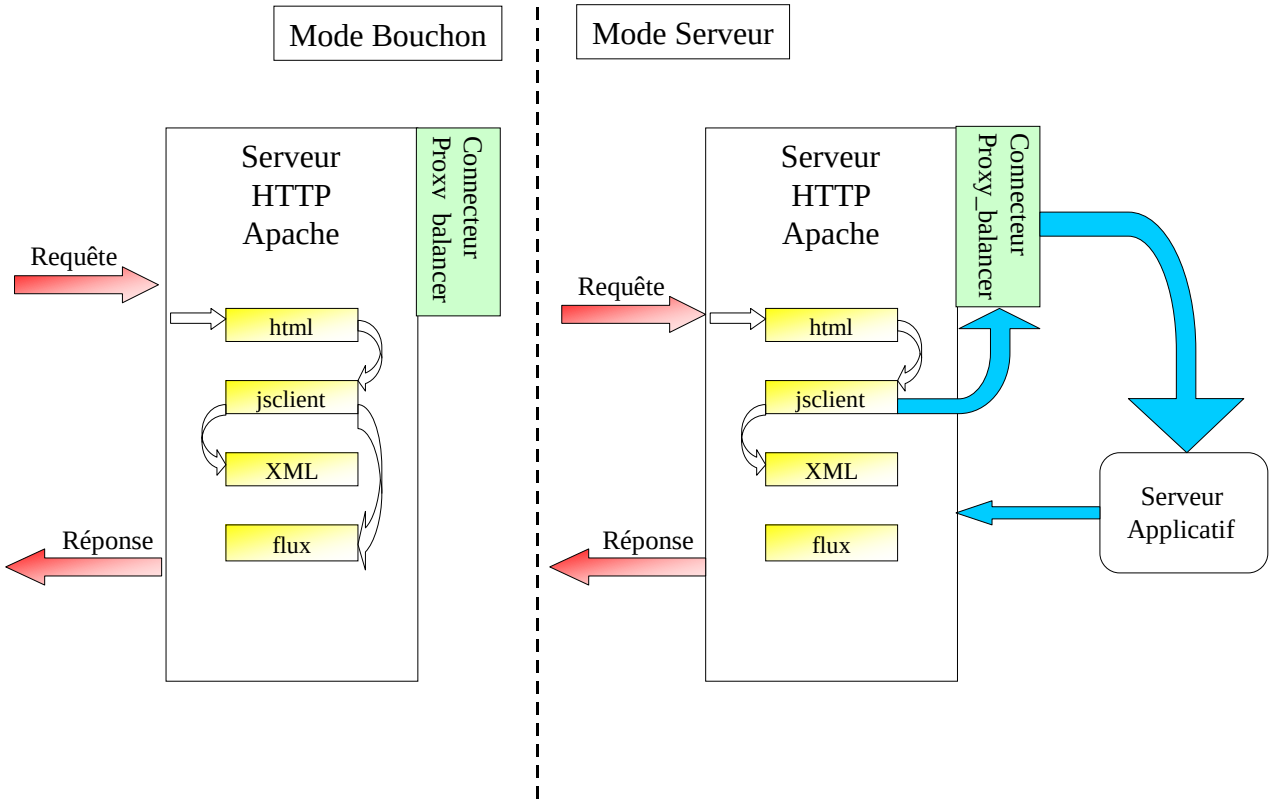
2.3 FLUX XML DYNAMIQUES

Les flux XML « dynamiques » (flux XML / PDF normalement issu du serveur applicatif) sont contenus dans des fichiers XML sous le répertoire <racine>/flux/.
La structure des répertoires contenus sous <racine>/flux/ doit respecter la structure des actions du serveur applicatif.
Exemple :

- Une action du serveur applicatif correspond au chemin /flux/protected/lp/actionA.xml
- Le répertoire /flux/ devra contenir un fichier de chemin /flux/protected/lp/actionA.xml

2.4 MODES DE FONCTIONNEMENT BOUCHON / SERVEUR

La maquette de l'application présenté au client doit pouvoir fonctionner en mode « bouchon » (i.e. sans la présence du serveur applicatif).
Les fichiers XML / PDF contenus sous le répertoire <racine>/flux/ permettent de « simuler » le fonctionnement du serveur applicatif, en contenant les flux XML / PDF qui sont normalement (en mode serveur) retourné par le serveur applicatif.



Le passage du mode « Bouchon » au mode « Serveur » se fait en activant / désactivant le module mod_proxy qui effectue une redirection des requêtes sur le répertoire <racine>/flux/ vers le serveur applicatif.

Pour permettre à cette architecture de fonctionner, il est important de respecter les points suivants :

1. Veiller à bien séparer les flux « statiques » des flux « dynamiques » lors du développement de la maquette
2. A chaque action du serveur applicatif (flux dynamique) doit correspondre un fichier XML / PDF contenant le même type de flux (la structure XML doit être la même) et possédant le même chemin que l'action correspondante

2.5 STRUCTURE D'UNE PAGE JAVASCRIPT

exemple.js

```
...
function preparePage()
{
 …
 XMLPageStatique = new parent.XMLObject(…,
                        ".../xml/... /xxx.xml", …);
 XMLPageStatique.importXML();
 …
}

function importerDonneesDynamiques() {
{
 …
 XMLPageDynamique = new parent.XMLObject(…,
                        ".../flux/protected/... /xxx.xml", …);
 XMLPageDynamique.importXML();
 …
}

function construireRecherche()
{
 …
 /*********************************/
 /* Gestion des données statiques */
 /*********************************/
 …
 /*********************************/
 /* Gestion des données dynamiques */
 /*********************************/
 …
```

① Importation du flux **XML statique** nécessaire à la construction de la page

② Importation du flux **XML dynamique** nécessaire à la construction de la page

③ Construction de la page à l'aide des flux XML importés

exemple.js

```
...

 composantRechercheLP.controlerSurface = function() {
   ...
 }

 ...

 composantRechercheLP.soumettre = function() {
   ...
   var retourAction = appelerAction(
                        ".../flux/protected/.../xxx.xml");
   ...
 }
```

④ Définition des contrôles de surfaces, appelés lors de la validation du formulaire

⑤ Définition de la fonction appelée lors de la validation du formulaire de la page

!!!
Tout appel à une action du serveur doit retourner un flux XML (de succès ou d'erreur): pas de gestion de la navigation côté serveur applicatif

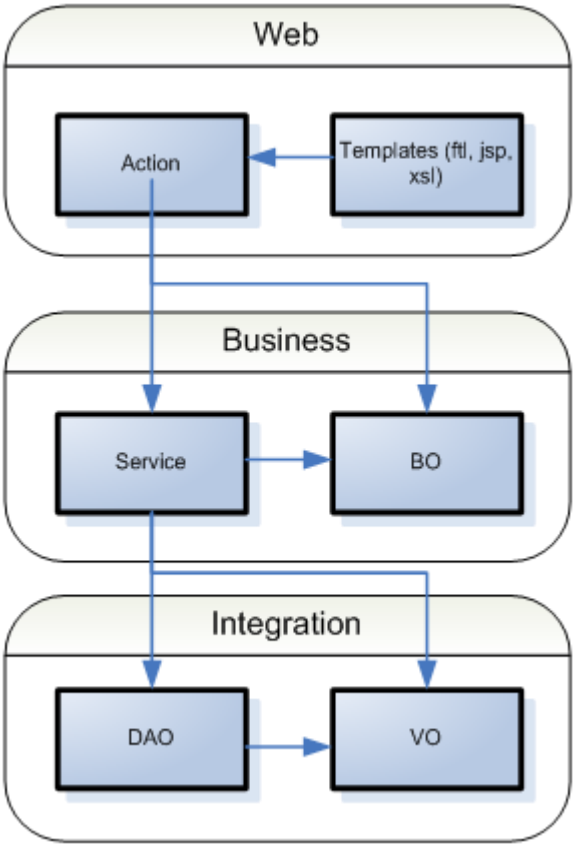
3 DEVELOPPEMENT SERVEUR

3.1 ARCHITECTURE APPLICATIVE

La partie serveur est implémentée pour la plateforme Java 5 et Java Enterprise Edition 1.4

3.1.1 DECOUPAGE

La partie serveur utilise une architecture en trois couches applicatives (3-Tiers) :



Chaque couche utilise des stéréotypes de classes qui lui sont propres.

Les contraintes structurant cette architecture sont :

1. Limiter la dépendance du code applicatif envers le socle technique, afin de réutiliser le même code dans des contextes d'exécutions différents (tests unitaires, mode bouchonné, …).
2. Avoir un couplage faible entre les composants applicatifs.
3. Permettre la réutilisation des services métiers dans d'autres applications.

3.1.1.1 COUCHE WEB

La couche web définit la partie Vue/Contrôleur du pattern MVC (Model-View-Controller). Elle s'organise autour des :

- Actions : Elles jouent le rôle de Contrôleur
- Templates : Ils sont en charges de fournir la Vue à la couche présentation

Pour plus d'informations consulter la section 3.3

3.1.1.2 COUCHE BUSINESS

La couche Business caractérise la logique métier de l'application. Elle se décompose en :

- Service : Les services se chargent d'effectuer les traitements sur les modèles.
- BO : Ils représentent le modèle métier de l'application.

Pour plus d'informations consulter la section 3.4

3.1.1.3 COUCHE INTEGRATION

Cette couche représente l'accès aux données et s'inscrit dans les principes du pattern DAO (Data Access Object). Elle contient les :

- DAO : Ces objets ont pour seul rôles d'accéder aux données
- VO : Ils constituent les objets de transfert entre le modèle métier et le modèle persistant. Leurs seuls objectifs est d'accueillir les données.

Pour plus d'informations consulter la section 3.5

3.1.2 FRAMEWORK

Les Framework open-source structurant cette architecture sont :

- **Struts2** : web-tiers (mapping de requêtes, validation serveur, i18n, redirection result)
- **Spring** :

  business-tiers : centralisation de la configuration dans un fichier XML, fabrique d'objets et résolutions des dépendances, intégration de différents Framework (dont Struts2 et IBatis)
- **IBatis** : mapping Objet/Relationnel

3.2 CREATION DU PROJET

Ce paragraphe décrit la procédure de création d'un projet Serveur sous Eclipse et partage dans subversion (SVN).

3.2.1 A PARTIR DU GABARIT

Suivre le document : A3_GUI_Creation_Rapide_Projet_ACube_J2EE_avec_LISE_v.3-1.0.pdf

3.2.2 A PARTIR DU PROJET EXEMPLE

3.2.2.1 INITIALISATION DU PROJET

1. Récupérer un modèle de projet. Deux solutions :
    a. Depuis SVN : ouvrir la perspective SVN, se positionner sur le module *EDS/Indus/A3/MAEE/sample*, puis cliquer sur « Export ». Renseigner le workspace d'eclipse comme chemin d'export. Cliquer sur OK.
    b. Depuis l'archive zip : Décompresser dans un répertoire du workspace d'eclipse.
2. Importer le projet : Cliquer sur File > Import > Existing Project Into Workspace, puis Next. Renseigner le chemin vers le répertoire sample du workspace. Cliquer sur OK.
3. Renommer le projet : Clic droit sur le projet importé > Refactor > Rename (ex :test)
4. Vérifier la compilation.
5. Sur le projet, clic droit > Properties > Web Project Settings : Renseigner le context root (rappel : si le contexte est « test », l'url de base pour accéder à Tomcat sera : http://localhost:8080/test)
6. Ajouter la vue Servers (Window > Show View > Other > Server > Servers
7. Dans la vue Servers, clic droit, New : choisir Apache Tomcat 5.5. Cliquer sur Next.
8. Si le runtime n'est pas configuré : indiquer le chemin vers le répertoire Tomcat 5.5 du poste de dév.
9. Sélectionner le projet créé dans la liste « available projects » et ajouter à la liste « Configured Projects ». Clic sur Finish

3.2.2.2 PARTAGE SVN

10. Sur le projet, clic droit > Team > Share Project : renseigner le nouveau répertoire SVN du projet (ex : test). Attention nécessite les droits en écriture sur le référentiel SVN

3.2.2.3 CREATION DE LA BASE DE TEST (MYSQL)

11. Installer le serveur MySQL (5) en local.
12. Se connecter en tant que root
13. Exécuter les scripts SGBD/schema.sql et SGBD/data.sql.
14. Créer le user « sample » (password sample) avec privilèges CRUD sur le schema sample.

3.2.2.4 LANCEMENT / TEST

15. Clic sur le serveur Tomcat et cliquer sur l'icône  pour le démarrer en mode debug.
16. Ouvrir le navigateur et taper l'url : http://localhost:8080/test, une page s'affiche avec une liste d'actions à tester

3.3 DEVELOPPEMENT PARTIE WEB

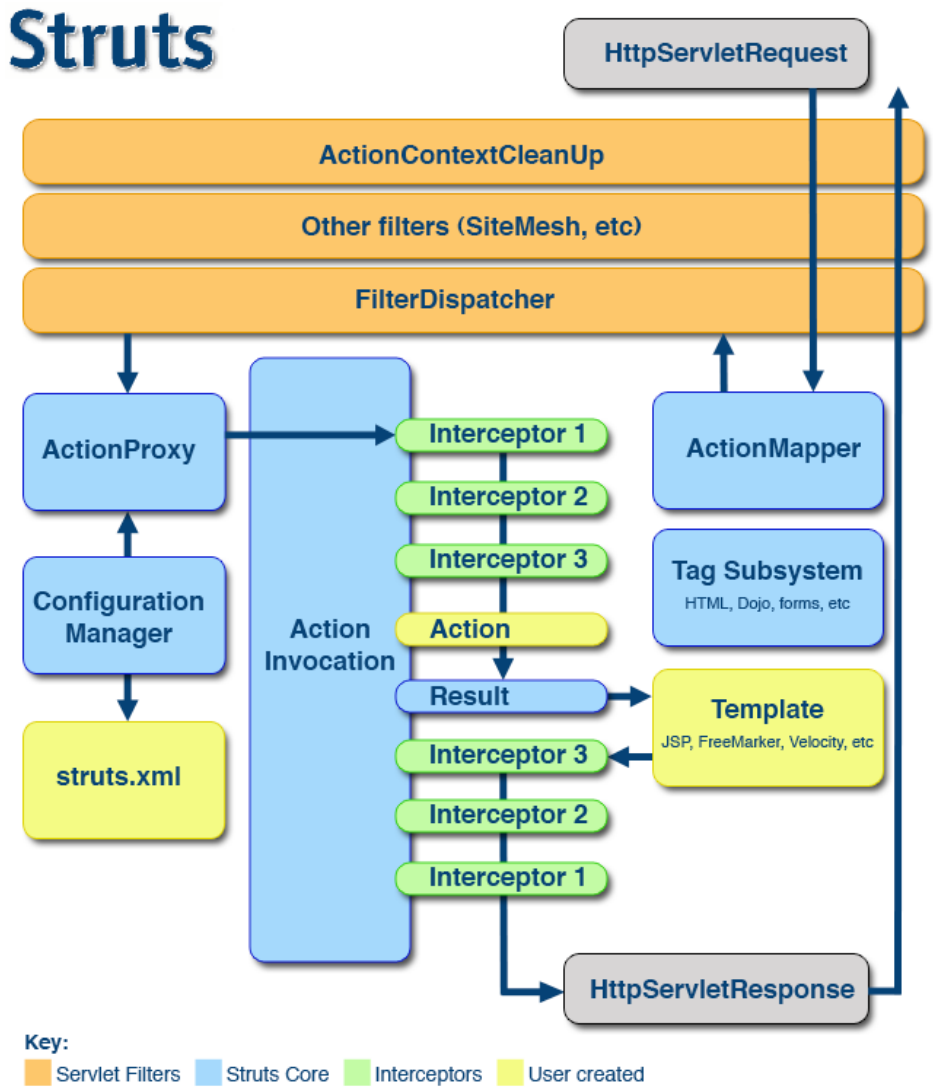
La partie Web est structurée autour du framework Struts 2 car il permet de répondre aux deux premières contraintes de l'architecture.

Struts 2 utilise les patterns Model-View-Controller (MVC) et Inversion Of Control (IOC).

-MVC permet d'améliorer la structuration et la séparation des responsabilités au sein de l'application Web. Le pattern MVC utilisé ici dispose d'un seul contrôleur qui intercepte toutes les actions utilisateur (c'est le MVC de type 2). Le contrôleur est paramétrable (via un ou plusieurs fichiers xml) et retransmet les requêtes à des classes dédiée (classes Action). Ces classes permettent d'agir sur le Modèle et informe le contrôleur sur l'état d'exécution. Le contrôleur donne ensuite la main à la vue qui permet de générer la réponse à renvoyer.

-IOC permet de supprimer les dépendances du code applicatif envers l'API Servlet, ce qui permet l'exécution des mêmes classes en dehors du conteneur Web (pour les tests unitaires par exemple)

3.3.1 ARCHITECTURE DE STRUTS 2



Etapes de traitement d'une requête http (http://localhost:8080/test/toto.xml) :

1. La requête arrive sur le conteneur de Servlet disposant du connecteur http sur le port 8080. Le conteneur redirige la requête vers le contexte « test » où l'application est déployée, et crée un objet Request et Response.
2. Ces objets traversent une chaine de filtres, notamment le FilterDispatcher de Struts.
3. Le filterDispatcher demande à l'ActionMapper si la requête doit être traitée par une action Struts (règle basée sur la forme de l'url, par défaut : *.xml, *.pdf, *.xls, *.doc)
4. Le FilterDispatcher demande à l'ActionProxy de retrouver l'action associée à l'uri « toto », via le fichier de configuration struts.xml
5. Une pile d'intercepteur (ici 1 à 3) est exécutée afin de fournir différents services communs aux actions.
6. L'action est exécutée.
7. Le contrôle est donné à un objet Result chargé de produire le flux de réponse HTTP. L'objet s'appuie sur un langage de Template.
8. Les intercepteurs sont dépilés.
9. Les filtres sont dépilés.
10. La requête revient au conteneur et ce dernier ferme le flux de réponse.

Note : Les intercepteurs permettent de factoriser des traitements communs autour des actions.
Ils peuvent s'exécuter :

- avant une action, par exemple pour initialiser des données sur certains types d'actions (implémentant les interfaces telles que PrincipalAware, ScopedModelDriven,…).
- Après une action et l'appel du Result
- Après une action mais avant l'appel du Result (PreResultListener)

Dans une application basée sur Struts, seules les classes Action (partie controller), les Templates (partie View) et le fichier de configuration seront à développer (cf. schéma d'architecture).

3.3.1.1 CONTROLLER

La partie « Controller » de l'architecture est implémentée par des classes « Action ».

Une classe Action permet de traiter une requête http, d'agir sur le Model et de déléguer la construction de la réponse à la Vue.

Une classe Action est un javabean possédant au moins une méthode de traitement. Par défaut Struts s'attend à trouver une méthode execute(), mais il est possible de spécifier une autre méthode dans la configuration du mapping. Une action peut donc déclarer plusieurs méthodes de traitement, afin de centraliser les opérations de type CRUD.

Les classes Action peuvent étendre la classe `com.opensymphony.xwork2.ActionSupport` afin de gérer la validation (Cf. exemple).

3.3.1.2 MODEL

La partie « Model » correspond aux entités métier de l'application (les BO). Il sera accédé via les classes de Service. La ou les classes de service seront mises à disposition de la classe Action par Spring via le constructeur (Cf. exemple).

3.3.1.3 VIEW

La partie « View » permet de créer un flux de réponse http en se basant sur le modèle des entités métier (BO), initialisés lors du traitement de l'action.

La vue invoquée par le contrôleur suite au traitement de l'action est paramétrée dans struts.xml (Cf paramétrage)

Afin de gagner en performance et en productivité, il est recommandé d'utiliser des templates JSP et non des templates XSLT pour la génération des flux XML, XLS, DOC et CSV.

3.3.2 PARTIE CONTROLLER : IMPLEMENTATION D'UNE CLASSE ACTION

- Dériver l'action de la classe ActionSupport, qui permet entre autres de bénéficier de la validation déclarative.
- Créer une propriété pour chaque paramètre de requête, en utilisant le type java compatible.
- Si une action utilise des services métiers, alors elle doit définir un constructeur ayant comme paramètres ces services. Ces derniers seront fournis par Spring à l'exécution.

  Attention : pas de getter et setter sur les services, sinon ils seront sérialisés en XML dans le cas d'une transformation XSLT.
- Créer éventuellement des méthodes d'exécution distinctes s'il s'agit d'une action de type CRUD sur un formulaire ou un tableau.

  Attention : les méthodes d'exécution ne doivent pas commencer par get, set ou is sinon elles seront sérialisées en XML, dans le cas d'une transformation XSLT.
- Créer des getter/setter sur les propriétés devant être accédés par les templates et les propriétés mappant les paramètres de requête.

Exemple :

```
/**
 * Gestion de la liste de produits
 *
 * @author EffiTIC
 */
public class GererProduitsAction extends ActionSupport {

    /**
     * Comment for <code>serialVersionUID</code>
     */
    private static final long serialVersionUID = 5563129040740608297L;

    /** La produit de présentation */
    protected Product product;

    /** La liste de présentation des produits */
    private ArrayList<Product> list;

    /** Le service métier des Produits */
    private ProductService productService;

    /*
     * Constructeur par défaut
```

```
     */
    public GererProduits(ProductService productService) {
        this.productService = productService;
        this.list = new ArrayList<Product>();
    }

    /*
     * Fournie la liste des produits
     * @return la liste des produits
     */
    public String list() {
        this.list.addAll(productService.list());
        return ActionSupport.SUCCESS;
    }

    /*
     * Ajoute un produit
     * @return SUCCESS
     */
    public String add() {
        productService.add(product);
        return ActionSupport.SUCCESS;
    }

    /*
     * Modifie un produit
     * @return SUCCESS
     */
    public String update() {
        productService.update(product);
        return ActionSupport.SUCCESS;
    }

    /*
     * Supprime un produit
     * @return SUCCESS
     */
    public String remove() {
        productService.remove(product);
        return ActionSupport.SUCCESS;
    }

    /*
     * Accesseur du produit
     * @return le produit
     */
    public Product getProduct() {
        return this.product;
    }

    /*
     * Modifieur du produit
     * @param product le produit à modifier
     */
```

```
    public void setProduct(
            Product product) {
        this.product = product;
    }

    /*
     * Accesseur de la liste de produits
     * @return la liste de produit
     */
    public ArrayList<Product> getList() {
        return this.list;
    }

    /*
     * Modifieur de la liste de produits
     * @param list la liste de produit à modifier
     */
    public void setList(ArrayList<Product> list) {
        this.list = list;
    }
}
```

Attention, la récupération des paramètres (post ou get) passés par Struts à l'action n'est rendue possible que si le nom du paramètre, déterminé par le client, correspond au nom de l'attribut de la classe action. Il est cependant possible d'utiliser des appellations différentes.

Pour cela, il faut utiliser les Alias de Struts, à configurer dans le fichier « struts.xml » au moment de la déclaration de chaque action :

```
<package name="commandes.gestionFournisseurs" namespace="/flux/protected/package"
extends="struts-acube-fwacubej2ee">
<action name="nomDeLAction"  class="acube.projet.web.action.nomAction"
method="nomMethode">
       <param name="aliases">#{
           'id'                : 'idFournisseur',
           'code'              : 'fournisseurBO.code',
           'libelle'           : 'fournisseurBO.libelle',
           'cadeaux'           : 'fournisseurBO.fournisseurDeCadeaux',
           'siren'             : 'fournisseurBO.siren',
           'adresse'           : 'fournisseurBO.adresse',
           'complementAdresse' : 'fournisseurBO.adresseComplementaire',
           'cp'                : 'fournisseurBO.codePostal',
           'ville'             : 'fournisseurBO.villeBO.libelle',
           'courriel'          : 'fournisseurBO.courriel'
        }
        </param>
        <result name="success" type="xslt">
             <param name="location">
               /templates/package/template.xsl
             </param>
        </result>
```

```
</action>
```

L'énumération des alias doit avoir la forme suivante :

'nom du paramètre passé par le client en post ou get' : 'Attribut dans l'action'.

On peut ainsi désigner dans l'action tout objet possédant les getters et setters correspondant. Il faut toutefois s'assurer que chaque objet a été préalablement instancié (dans le constructeur de la classe action, par exemple).

3.3.3 PARTIE VUE : IMPLEMENTATION D'UN TEMPLATE

Les templates JSP et XSL sont à définir dans le répertoire WebContent/templates

Exemple de template JSP :

```
<?xml version="1.0" encoding="UTF-8"?>
<%@ page contentType="text/xml; charset=UTF-8"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<ARTICLES>
      <s:iterator value="list">
            <ARTICLE>
                  <ID><s:property value="productId"/></ID>
                  <LABEL><s:property value="productLabel"/></LABEL>
                  <PRICE><s:property value="productPrice"/></PRICE>
                  <DATE><s:property value="productValidity"/></DATE>
            </ARTICLE>
      </s:iterator>
</ARTICLES>
```

Attention : Ne pas oublier le contentType (généralement text/xml) !

Les données spécifiées dans les tags de template (JSP ou XSLT) sont recherchées :

- dans le pageContext (variables créées dans la JSP)
- dans les propriétés de l'action
- dans les attributs de requête (requestScope)
- dans les attributs de session (sessionScope)
- dans les attributs d'application (applicationScope)

Si le contexte n'est pas spécifié, Struts effectue une recherche dans cet ordre.

3.3.3.1 INCLUSION / IMPORT DE XSL :

```
<xsl:include href="response:/templates/com/CommonMethod.xsl" />
<xsl:import href="response:/templates/com/CommonMethod.xsl" />
```

3.3.3.2 CONTENT-TYPE XSL

Struts 2 se base sur l'attribut media-type pour créer le content-type du flux de réponse http. Par exemple pour générer un document excel il faut avoir:

```
 <xsl:output method="html" encoding="iso-8859-1" media-type="application/excel" />
```

3.3.4 NORMES POUR JSP

3.3.4.1 COMPILATION DES JSP

Toutes les JSP doivent compiler.

Les fragments de JSP inclus à la compilation via le tag <%@ include file…> doivent être suffixés

.jsp si le fichier correspondant compile;

.inc sinon

3.3.4.2 ENTETE JSP ET JEU DE CARACTERES

Le jeu de caractère UTF-8 doit être positionné dans l'entête de chaque jsp (pas de factorisation dans une jsp incluse) :

```
<%@ page language="java" contentType="text/xml; charset=UTF-8" pageEncoding="UTF-8"%>
```

- L'attribut contentType correspond au type de contenu envoyé au navigateur via un header http,
- L'attribut pageEncoding correspond au jeu de caractère de la jsp elle-même

3.3.4.3 ESCAPE & ENCODAGE

Les chaînes de caractère doivent être converties en version compatible XML (remplacement > par > …).

On peut effectuer cette conversion de deux manières :

- Pour les jsp utilisant directement la jstl, il existe deux façons de procéder :

une expression EL du type

```
${x.y}
```

doit être remplacée par un code du type

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
…
<c:out value="${x.y}"/>
```

ou encore

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsp/jstl/functions"%>
…
${f:escapeXml(x.y)}
```

- Pour les jsp struts, le tag à utiliser est s:property :

le code de remplacement est

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

```
…
<s:property value="x.y"/>
```

3.3.4.4 PROLOGUE XML ET CONTENTTYPE

Dans le cas d'une génération d'un document XML par une jsp, les contentType jsp et xml doivent être les mêmes :

```
<?xml version="1.0" encoding="UTF-8"?>
<%@ page language="java" contentType="text/xml; charset=UTF-8" pageEncoding="UTF-8"%>
```

3.3.5 GESTION DU CACHE CLIENT

Il est possible coté client de spécifier à la génération du livrable un préfixe sur les dossiers du client pour paramétrer le cache dans Apache. Ainsi les répertoires sont renommés et les urls changent.

Jsclient devient jsclient-v1.0.

Cette fonctionnalité coté client est gérée par la tâche ANT.

Coté serveur, il est possible de gérer cette fonctionnalité dynamiquement.

3.3.5.1 POUR LES JSP

Pour spécifier le préfixe, il faut définir une variable dans le fichier web.xml :

```
    <context-param>
        <param-name>tagDir</param-name>
        <param-value>-2.10</param-value>
    </context-param>
```

Ainsi la variable « tagDir » sera disponible dans les JSP et XSL.

Dans les JSP utiliser le tag ci-dessous :

```
<%=request.getContextPath() %><s:url value="%{#application.tagDir}/flux/Logout.xml" />
```

3.3.5.2 POUR LES FEUILLES XSL

Dans l'action implémenter l'interface « TagDirAware » comme l'exemple ci-dessous :

```
public String getTagDir() {
    return ServletActionContext.getServletContext().getInitParameter("tagDir");
}
```

Dans la feuille XSL, la propriété « tagDir » sera disponible :

```
<xsl:value-of select="tagDir" />
```

Cette méthode fonctionne aussi avec les JSP.

3.3.6 CONFIGURATION DU CONTROLLER

Un paramétrage est nécessaire à Struts pour mettre en correspondance les éléments de l'architecture MVC.

Ce paramétrage est défini dans le fichier **struts.xml**. Ce fichier doit être présent dans le classpath de l'application. Ici il est placé à la racine du répertoire des sources (src)

Il sera possible de découper ce paramétrage en plusieurs fichiers si l'application devient importante. Dans ce cas, struts.xml deviendra le fichier maitre et le nommage des fichiers fils se fera avec la convention struts-mondomaine.xml.

3.3.6.1 PRINCIPE

Le paramétrage consiste principalement à mapper des URL sur des actions java et pour chaque action, de définir les vues selon le type de résultat produit par l'action.

Le paramétrage est structuré : chaque action appartient à un package qui représente une partie de l'URL. De plus, chaque action possède un alias qui représente une autre partie de l'url.

Ainsi l'url suivante :

```
http:// domaine :port/contexte/flux/protected/domaine/sousdomaine/monAliasAction.xml
```

Sera mappée par :

```
<package name="domaine" namespace="/flux/protected/domaine/sousdomaine"
extends="acube-default">
      <action name="monAliasAction">
            <result name="success">
                WEB-INF/jsp/domaine/monAction.jsp
            </result>
        </action>
</package>
```

En cas de succès, le traitement de la requête est transmis à un result, ici de type JSP, afin de construire la réponse sous forme d'un flux XML.

En cas d'erreur de validation (« input ») ou fonctionnelle (« error »), Cf Gestion des erreurs

Il existe plusieurs type de result : JSP, Redirect, RedirectAction, etc. Voir :
http://struts.apache.org/2.x/docs/result-types.html

Note : Il est possible de ne pas spécifier de vue « success », dans ce cas Struts utilisera le Template par défaut « WEB-INF/templates/succesVO.jsp)

3.3.6.2 MAPPING DES METHODES D'EXECUTION

Il est possible d'apporter une structuration supplémentaire en regroupant plusieurs méthodes d'exécution dans une action (par exemple le chargement et l'enregistrement d'un formulaire).

Ainsi l'url suivante :

```
http://domaine:port/contexte/flux/protected/domaine/monAliasAction-maMethode.xml
```

Sera mappée par :

```
<package name="domaine" namespace="/flux/protected/domaine" extends="acube-
default">
<action name="monAliasAction-maMethode" method="maMethode"
class=”fr.acube.web.action.MonAliasAction”>
                <result name="success">
                 WEB-INF/jsp/domaine/monAction.jsp
                </result>
        </action>
</package>
```

3.3.6.3 WILDCARD-MAPPING

Si une classe d'action regroupe beaucoup de méthodes d'exécution (type CRUD), alors pour ne pas surcharger la configuration il sera utile d'utiliser le Wildcard-Mapping :

```
<package name="domaine" namespace="/flux/protected/domaine" extends="acube-
default">
   <action name="monAliasAction-*" name="{1}">
        <result name="success">
                WEB-INF/jsp/domaine/{1}.jsp
        </result>
   </action>
</package>
```

Attention *: ne pas utiliser le Dynamic Method Invocation (DMI), qui consiste à appeler la méthode d'exécution directement depuis l'url sans passer par le mapping (avec le !) :*

http://localhost:8080/test/flux/protected/action.xml!methode

3.3.6.4 CONFIGURATION DES AUTORISATIONS

But : limiter l'accès aux actions suivant les rôles J2EE de l'utilisateur (obtenus via l'opération de login)

Exemple :

```
<package name="domaine" namespace="/flux/protected/domaine" extends="acube-
default">
   <action name="monAliasAction">
        <interceptor-ref names="acube-stack">
                <param name="acube-security.allowedRoles">admin,member</param>
        </interceptor-ref>
        <result name="success">
                WEB-INF/jsp/domaine/action.jsp
        </result>
   </action>
```

```
</package>
```

Dans ce cas seuls les rôles admin et member auront accès à l'action.

Une configuration « négative » peut être utilisée en remplaçant « allowedRoles » par « disallowedRoles ».

La configuration peut également se faire pour un package entier (et peut être aussi surchargée par certaines actions) :

```
<package name="domaine" namespace="/flux/protected/domaine" extends="acube-
default">
        <interceptors>
            <interceptor-stack name="acube-stack">
                    <interceptor-ref name="acube-stack">
                        <param name="acube-security.allowedRoles">
                            admin
                        </param>
                    </interceptor-ref>
            </interceptor-stack>
        </interceptors>
</package>
```

3.3.6.5 RAPPEL SUR LA GESTION DE LA NAVIGATION

La gestion de la navigation entre les différentes pages HTML se fait du côté client riche : aucune gestion de la navigation ne doit être faite côté serveur applicatif (via des redirections).

Exception : la redirection vers la page de login si on n'est pas authentifié, la redirection vers la page d'index et la redirection après déconnexion. Dans ce cas, struts 2 propose un result de type redirection.

3.3.6.6 EXTENSION DU PARAMETRAGE PAR DEFAUT

Le Framework LISE étend la pile par défaut de struts (default-stack) en une pile « acube-stack ». Cette pile défini un comportement par défaut qu'il est possible d'étendre à nouveau selon les besoins du projet.

Intercepteurs spécifique LISE:

-**SecurityInterceptor** : Extension de RolexInterceptor de struts2 afin de paramétrer les roles autorisés par action dans le fichier struts.xml (tel que sur l'exemple vue en 3.3.6.4).

-**CacheControlInterceptor** : ajout du contrôle du cache navigateur (pas de mise en cache + compatibilité https avec IE)

Surchage de paramétrage :

-Redirection des exceptions vers ErrorReportAction

Note : La pile « acube-stack » sera utilisée automatiquement si le package d'action étends « acube-default » plutôt que « struts-default »

3.3.7 GESTION DE LA VALIDATION

Struts propose trois niveaux de validation :

1. Validation automatique des types des paramètres de requête en fonction des propriétés de l'action.

2. Exécution de règles sur les champs de formulaire définies dans un fichier XML (validation déclarative Struts-Validator)
3. Exécution de la méthode « validate() » dans l'action .

> _La méthode de traitement de l'action (execute() par défaut) est exécutée une fois que tous les contrôles sont effectués._

Prérequis :

L'action doit dériver de **com.opensymphony.xwork2.ActionSupport**

3.3.7.1.1 Validation des types de champs

Les paramètres de requêtes sont mappés sur des propriétés java typées de l'action (String, Integer, BigDecimal, Date, …).

Si une conversion est impossible (type non conforme), un message d'erreur est ajouté à la liste ActionErrors de la classe ActionSupport. Le message est retourné au client sous forme d'un flux XML (ExceptionInterceptor)

Il est possible de redéfinir le message dans le fichier com\opensymphony\xwork2\xwork-messages_fr.properties

3.3.7.1.2 Validation déclarative XML

- Créer un fichier *MonAction-validation.xml* dans le package acube.projet.web.action.

Ce fichier sera valable pour toutes les méthodes de traitement (méthode execute() par défaut).

Si la validation est propre à une méthode d'exécution, par exemple update() :

- créer un fichier *MonAction-update-validation.xml* dans le package *acube.projet.web.action*

Les validators disponibles, ainsi que la syntaxe XML sont décrits ici :

Z:\UsineA3\Compétence\DocumentationTechnique\strut2\validation.html

Règle : utiliser les field-validator pour les contrôles par champ, et les non-field-validator pour les contrôles interchamps.

Règle : utiliser au maximum la validation XML, notamment les expression OGNL, pour tous les contrôles n'effectuant pas d'accès à la base de donnée.

3.3.7.1.3 Validation JAVA

- Surcharger la méthode validate() de la classe ActionSupport

Règle : utiliser la validation JAVA uniquement pour tous les contrôles nécessitant un accès à la couche Service.

3.3.8 GESTION DE LA SESSION

3.3.8.1 PRINCIPE

La session http est une zone de mémoire serveur allouée à un utilisateur (pour un navigateur web). Elle permet :

- De conserver les données entres les différentes actions utilisateur (ex : entre les pages d'un assistant de saisie)
- De faire office de cache : on charge les données en début de cas d'utilisation seulement, afin d'optimiser les accès au système de persistance.
- Rappel : lors de la déconnexion de l'utilisateur, les données sont perdues et la mémoire utilisée est rendue disponible (sauf si on y stocke des objets du contexte serveur ou si on s'amuse à y stocker des threads (interdit !) : ces derniers ne seront pas libérés)

IMPORTANT : S'assurer de supprimer les données à la fin du cas d'utilisation, afin d'éviter les effets de bords (pouvoir rejouer le cas d'utilisation de manière identique)

3.3.8.2 SESSIONAWARE

Pour qu'une action accède à la session il suffit d'implémenter l'interface SessionAware. On récupère une map contenant les attributs de session sous forme de (clé, valeur)

```
public class CommandeAction extends ActionSupport implements SessionAware {
        Map session;
        public void setSession(Map session) {
                this.session = session;
        }
}
```

Localisation des clés de session :

Les clés de session peuvent être placées dans l'action elle-même si la session sert de partage entre plusieurs méthodes d'exécution, ou dans une classe commune à l'ensemble des cas d'utilisation.

Par contre il faut **Eviter une classe commune à l'application !**

3.3.8.3 SCOPE INTERCEPTOR

But : Permet de simplifier les enchainements d'actions de type Assistant de saisie.

Cf. http://struts.apache.org/2.0.11/docs/scope-interceptor.html

Exemple :

```
<action name="page1" class="acube.projet.web.action.AssistantAction1">
        <result name="success">/jsp/assistant.jsp</result>
        <interceptor-ref name="basicStack"/>
        <interceptor-ref name="scope">
                <param name="key">CLASS</param>
                <param name="session">mySharedBO</param>
                <param name="type">start</param>
        </interceptor-ref>
```

```
</action>
<action name=" page2" class=" acube.projet.web.action.AssistantAction2">
         <result name="success">/jsp/assistant.jsp</result>
         <interceptor-ref name="scope">
                  <param name="key">CLASS</param>
                  <param name="session">mySharedBO</param>
                  <param name="type">end</param>
         </interceptor-ref>
         <interceptor-ref name="basicStack"/>
</action>
```

NB : on peut aussi gérer les pages dans une seule action comportant plusieurs méthodes de traitement.

3.3.8.4 MODELE DU FORMULAIRE (SCOPEDMODELDRIVEN) :

Struts 1.x nécessitait l'utilisation d'un objet « ActionForm » pour modéliser les données du formulaire. Cet objet pouvait être stocké en requête ou en session.

Struts 2 permet maintenant de mapper les données du formulaire directement dans l'action, Mais l'ancien mécanisme peut être reproduit avec l'intercepteur `ScopedModelDrivenInterceptor`. Les actions qui souhaitent passer par un objet ActionForm intermédiaire doivent implémenter l'interface ScopedModelDriven<ClasseActionForm>, où la classe « ClasseActionForm» est un simple bean Java sérialisable. L'intercepteur possède comme attribut le scope égal à « request » (par défaut) ou « session ».

Pour les actions utilisant cet intercepteur, les paramètres de requêtes seront alors mappés par défaut sur l'ActionForm et non l'Action. Et l'ActionForm pourra être partagée automatiquement entre plusieurs actions.

Il est généralement plus simple de mapper les paramètres de requête sur un ou plusieurs objets BO référencés dans l'action, qu'on stocke ensuite en session grâce à SessionAware.

3.3.8.5 SYNCHRONISATION SESSION / BASE

Entre le début et la fin d'un cas d'utilisation, les données de session peuvent être périmées (à cause des accès concurrents au système de persistance).
Deux approches selon le niveau de concurrence d'accès aux données :

- on ne met à jour que les données modifiées par l'utilisateur ou, ce qui est équivalent, on relit les données, on merge avec les données modifiées, puis on met à jour dans la même transaction
- on met en place un système de versionning (principe de CVS ou SVN)

3.3.9 UPLOAD / DOWNLOAD DE FICHIERS

3.3.9.1 UPLOAD

Coté client : le formulaire doit être de type multipart et comporter au moins un champ input de type file

```
<form enctype="multipart/form-data" action="monaction.xml" method="post">
        …
        Fichier 1 : <input type="file" name="myDoc" />
        Fichier 2 : <input type="file" name="myDoc" />
        …
```

```
</form>
```

Côté serveur : l'action doit implémenter ces méthodes :

```
public void setMyDoc(File[] myDocs)
public void setMyDocContentType(String[] contentTypes)
public void setMyDocFileName(String[] fileNames)
```

NB : les crochets [] sont facultatifs si un seul fichier est uploadé

Explication : Struts 2 utilise commons-fileupload pour parser les requêtes multipart.

Les fichiers uploadés sont enregistrés temporairement sur disque (nom uniques par session). L'accès au flux de données se fait via les objets de type java.io.File. Les contentType et le fileName correspondent au type et au nom de fichier coté client.

Pour accéder au flux de donnée en JAVA :

```
InputStream is = new BufferedInputStream(new FileInputStream(myDocs[0]));
try{
   //… traitement…
} finally {
   is.close(); // et surtout on n'oublie pas de le fermer!
}
```

Pour le récupérer le flux dans d'un tableau de byte :

```
byte[] tab = new byte[is.available()] ;
is.read(tab);
```

Paramétrage par défaut (modifiable dans le struts.xml):

Propriété	Description	Valeur par défaut
struts.multipart.parser	Parser de requêtes multipart	Jakarta (Commons FileUpload)
struts.multipart.saveDir	Répertoire temporaire pour les fichiers uploadés	javax.servlet.context.tempdir tel que défini par Tomcat
struts.multipart.maxSize	Quantité uploadée en octet. Si plusieurs fichiers, s'applique au total des fichiers	2 Mo

3.3.9.2 DOWNLOAD

Classe Action :

```
byte[] data;
…
this.setContentDisposition("filename=\"monFichier.pdf\"");
this.setContentType("application/pdf");
this.setInputStream(new ByteArrayInputStream(data));
```

```
return SUCCESS;
```

Struts.xml :

```
<action name="export" class="acube.projet.web.action.Exporter">
        <result name="success" type="stream">
                <param name="contentType">${contentType}</param>
                <param name="inputName">inputStream</param>
                <param name="contentDisposition">
                        ${contentDiposition}
                </param>
                <param name="bufferSize">1024</param>
        </result>
</action>
```

3.3.10 INTEGRATION AVEC SPRING

Struts doit être paramétré pour déléguer la construction des actions à Spring (c'est le cas pour l'application d'exemple). Cela permet à Spring de fournir les instances de Services aux actions Struts.

3.3.11 UTILISATION DES SERVICES SPRING DANS STRUTS2

3.3.11.1 UTILISATION HORS COUCHE SERVICE ET PRESENTATION

Il peut être nécessaire d'utiliser un service spring hors de la couche service et présentation.

Ceci se fait en deux étapes :

1. Récupérer ou instanciation d'une factory Spring
2. Instanciation du service par la factory Spring.

L'étape 1 doit être faite à l'initialisation de l'application (ou du composant).

L'étape 2 doit être faite lors du traitement.

Le pseudo code pour l'étape 1 est :

```
import org.springframework.context.support.ClassPathXmlApplicationContext
…
ClassPathXmlApplicationContext serviceFactory =
 new ClassPathXmlApplicationContext (
new String [] {"WEB-INF/appContext.xml", "WEB-INF/appContext-dao.xml"});
```

Le pseudo code pour l'étape 2 est :

```
…
```

```
MaClasseDeService monService = serviceFactory.getBean(« monService ») ;
…
```

Cette méthode est à proscrire dans le cas d'une application web (cf. § suivant)

3.3.11.2 UTILISATION DANS LA COUCHE PRESENTATION

La factory utilisée par struts2 est créée par le ServletContextListener fourni par Spring.

```
org.springframework.web.context.ContextLoaderListener
```

Les fichiers de configuration Spring sont indiqués par les nœuds suivants dans le web.xml

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath*:appContext.xml classpath*:appContext-security.xml
classpath*:appContext-dao.xml</param-value>
</context-param>
```

Le ContextLoaderListener met le ApplicationContext Spring à disposition de l'application sous forme d'attribut du servletContext.
Le nom de l'attribut est
org.springframework.web.context.WebApplicationContext. *ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE*
Le pseudo code pour récupérer la factory est :

```
ApplicationContext springServiceFactory =(ApplicationContext)
getServletContext().getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEX
T_ATTRIBUTE);
```

3.3.12 INTEGRATION DANS JEE

Reprendre le fichier web.xml de l'application d'exemple pour la définition du paramétrage de Struts 2.

```
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
            org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
        <init-param>
            <param-name>config</param-name>
            <param-value>struts.xml</param-value>
        </init-param>
    </filter>
```

3.3.13 TYPES JAVA SPECIFIQUE

Il est possible que certains objets métier de la partie Java utilisent des types Java spécifique. Par exemple on peut imaginer une classe qui représenterait une date avec des règles de gestion particulière et une structure particulière. Il est possible dans Struts2 de définir des convertisseurs particulier pour une classe données afin de faire le mapping d'une part entre le champ et la vue mais aussi entre les paramètres de la requête et l'objet java. Pour cela, il faut créer une classe qui étend la classe StrutsTypeConverter.

Exemple :

```
public class DateIncompleteConverter extends StrutsTypeConverter {

    /** {@inheritDoc} */
    @SuppressWarnings("unchecked")
    @Override
    public Object convertFromString(
            Map arg0, String arg1[], Class arg2) {

        DateIncomplete dateIncomplete = new DateIncomplete();
        dateIncomplete.setString(arg1[0]);

        return dateIncomplete;
    }

    /** {@inheritDoc} */
    @SuppressWarnings("unchecked")
    @Override
    public String convertToString(
            Map arg0, Object arg1) {

        return arg1.toString();
    }

}
```

Cette classe contient deux méthodes :

- convertFromString : qui permet de faire la conversion des paramètres de la requête en l'objet que l'on souhaite (ici DateIncomplete)
- convertToString : qui permet de définir le format de sortie pour la vue. (Exemple : sortie du s:property dans les JSP)

Ensuite il suffit de déclarer à Struts2 que pour le type DateIncomplete, il faut utiliser le converter DateIncompleteConverter. Ceci s'effectue dans un fichier xwork-conversion.properties comme ceci :

```
# syntax: <type> = <converterClassName>
acube.projet.utility.DateIncomplete=acube.projet.web.action.utility.DateIncomplete
Converter
```

3.4 DEVELOPPEMENT PARTIE BUSINESS

3.4.1 SERVICES

Les Services représentent la logique métier de l'application. Ils doivent être réutilisables au sein de l'application (partage entre domaines fonctionnels), mais également entre applications (au travers d'une couche Web-Service par exemple).

- Les services sont implémentés avec des classes JAVA simple. Ils possèdent obligatoirement une interface JAVA.

- Les services peuvent effectués des traitements simple ou complexe en réutilisant d'autres services (internes à l'application, ou externe).

- Les services collaborent avec les autres services uniquement via leur interface. C'est Spring qui est chargé de fournir l'implémentation appropriée

- Les services collaborent avec les DAO uniquement via leur interface. C'est Spring qui est chargé de fournir l'implémentation appropriée

- Les services sont généralement des singletons. Ils ne doivent donc pas contenir de données membres constituant un état sinon celles-ci conserveront leurs valeurs d'initialisation d'un appel à l'autre (voir exemple).

- Les services sont décrits dans le fichier Spring **src/appContext.xml**

Exemple de service contenant une donnée membre qui constitue un état :

```
public class MonServiceImpl {

    /** <code> maListeErreurs </code> the maListeErreurs     */
    private List<HistoriqueBO> maListeErreurs;

    /** <code>daoHistorique</code> the daoHistorique     */
    private HistoriqueDAO daoHistorique;

    /** <code>daoPurge</code> the daoPurge     */
    private PurgeDAO daoPurge;

    /**
     * @param id identifiant d'une boite aux lettres
     * @return la liste des historiques
     */
    public List<HistoriqueBO> doTraitement(String id) {
        List <HistoriqueBO> liste = new ArrayList<HistoriqueBO>();
        HistoriqueBO historique = new HistoriqueBO();
        historique.setIdBal(id);
        liste = this.daoHistorique.selectHistoriqueByIdBal(
                                historique.getHistoriqueVO());
        for (HistoriqueBO histo : liste) {
            if (histo.getHisAction() == null) {
```

```
                //Ajout d'elements à un objet membre d'un singleton
                this.maListeErreurs.add(histo);
            }
        }
        liste.removeAll(this.maListeErreurs);
        return liste;
    }

    /**
     * @return liste des purges
     */
    public List<Purge> doSomethingElse() {
        //utilisation de la liste du singleton dont le contenu peut être modifié
        //au moment de sa lecture par simple appel à la méthode doTraitement
        for (HistoriqueBO histo : this.maListeErreurs) {
            Purge maPurge = new Purge();
            maPurge.setPurIdbal(histo.getIdBal());
            maPurge.setPurDatepurge(new Date());
            this.daoPurge.insert(maPurge);
        }
        PurgeExample example = new PurgeExample();
        example.setOrderByClause("DATE_PURGE");
        return this.daoPurge.selectByExample(example);
    }
}
```

Remarque : Pour cette exemple il est conseillé de supprimer la donnée membre `maListeErreurs` du service et de passer cet objet liste en paramètre des fonctions `doTraitement` et `doSomethingElse`.

Exemple de déclaration de service utilisant un autre service :

```
<bean id="EntrepriseService"
class="acube.projet.business.service.EntrepriseServiceImpl">
                <constructor-arg ref="EntrepriseDAO"/>
                <constructor-arg ref="ReferenceService"/>
</bean>

<bean id="ReferenceService"
class="acube.projet.business.service.ReferenceServiceImpl">
        <constructor-arg ref="ReferenceDAO"/>
</bean>
```

3.4.2 BUSINESS OBJETS (BO)

Les BO représentent le modèle métier de l'application.

- Un BO est une classe JAVA simple sans dépendance sur l'architecture technique. Il possède au moins un constructeur par défaut et chaque propriété est accessible via des getter/setter.

- Un BO est constitué de propriétés simples, mais peut aussi étendre ou agréger d'autres BO.

- Un BO peut être construit à partir d'un unique VO (mapping un vers un) ou d'un ensemble de VO provenant éventuellement de sources différentes (mapping un à plusieurs). La conversion entre BO et VO sera toujours effectuée dans la couche service.

- Un BO peut contenir de la logique métier, mais ne doit pas faire appel à la couche service, dao, accéder à d'autres ressources, …

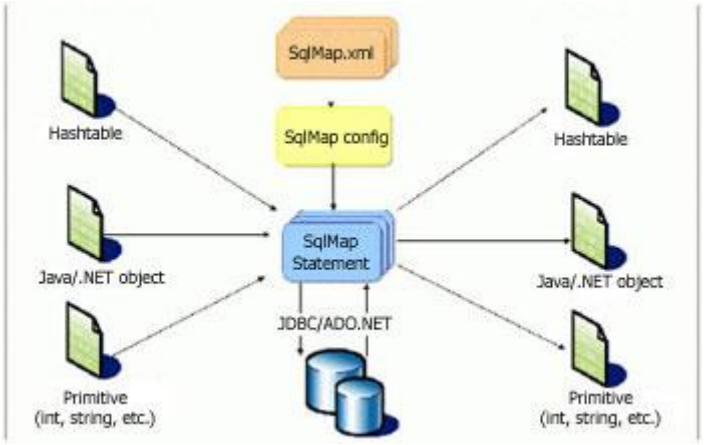
3.5 DEVELOPPEMENT PARTIE INTEGRATION

3.5.1ARCHITECTURE

Le framework IBatis est composé de deux modules : IBatis-Data Mapper et IBatis–Dao.

Cependant l'architecture Acube étant fortement basée sur Spring, c'est SpringDao avec support Ibatis qui sera utilisé en remplacement de IBatis-DAO, afin notamment de bénéficier de la gestion des transactions.

Data-Mapper est le module de mapping entre les requêtes SQL et les javabeans (VO ou BO). Il permet de mapper les types de base du langage (types primitifs, wrapper, String, Date,…) sur les types JDBC, mais également de mapper un resultset sur un graphe d'objets.



3.5.2 GENERATEUR IBATOR

Les DAO et VO peuvent être générés avec Ibator, afin de disposer d'un jeu de méthodes CRUD de départ. Le code pourra ensuite être personnalisé et régénéré sans perdre ses modifications. Attention : les méthodes CRUD générées n'opèrent que sur une table.

Pré-requis :

- Disposer du plugin Ibator for eclipse (faire un update sur le serveur d'indus A3, si besoin)
- Avoir créé le schéma de la base.

Etapes :

- Créer un fichier abatorConfig.xml à la racine du projet contenant au minimum :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE abatorConfiguration PUBLIC "-//Apache Software Foundation//DTD Abator
for iBATIS Configuration 1.0//EN"
  "http://ibatis.apache.org/dtd/abator-config_1_0.dtd">

<abatorConfiguration>
  <abatorContext generatorSet="Java5">    <!-- TODO: Add Database Connection
Information -->
    <jdbcConnection driverClass="com.mysql.jdbc.Driver"
        connectionURL="jdbc:mysql://130.177.222.200:3308/gca3_dev"
        userId="gca3_dev_dbo"
        password="gca3_dev_dbo">
      <classPathEntry location="D:\Documents and
Settings\pzh8z1\workspace\GCA3\Serveur\WEB-INF\lib\mysql-connector-java-3.1.7-
bin.jar" />
    </jdbcConnection>

    <javaModelGenerator targetPackage="com.eds.gca3.integration.vo"
targetProject="GCA3\Serveur\src" />
    <sqlMapGenerator targetPackage="com.eds.gca3.integration.dao.map"
targetProject="GCA3\Serveur\src" />
    <daoGenerator type="SPRING" targetPackage="com.eds.gca3.integration.dao"
targetProject="GCA3\Serveur\src" />

    <table schema="dbo" tableName="%">
    </table>

  </abatorContext>
</abatorConfiguration>
```

- Configurer la connexion JDBC (le chemin du driver est absolu)
- Dans les balises javaModelGenerator, sqlMapGenerator, daoGenerator , attribut targetPackage, remplacer le package « com.eds.gca3 » par celui du projet. Dans l'attribut targetProject, remplacer « GCA3 » par le nom du projet Eclipse
- Enfin pour lancer la génération : clic droit sur abatorConfig.xml / Generate IBATIS Artifacts

Fichiers générés :

Liste des fichiers générés pour une table « MA_TABLE » du schéma :

Artefact généré	**fonction**
integration.dao.MaTableDAO	interface du DAO
integration.dao.MaTableDAOImpl	implémentation du DAO pour Spring/IBatis
integration.dao.map.ma_table_sqlMap.xml	Requêtes SQL
integration.vo.MaTable	Classe java permettant de mapper chaque colonne d'une table, excepté les blobs. La conversion de noms et de types SGBD vers Java est définies par défaut dans Abator, mais surchargeable.
integration.vo.MaTableWithBLOBS	Classe java permettant de mapper chaque colonne d'une table, blob ou clob inclus.
integration.vo.MaTableExample	Classe java permettant de créer des filtres de recherche/maj via le DAO (pattern « query by example »)
Integration.vo.MaTableKeys	Classe représentant la clé de la table si la clé est composite

3.5.2.1 GESTION DES CLOB ET BLOB

Les types SGBD CLOB et BLOB sont mappés respectivement sur les types « java.lang.String » et « byte[] »

Pour les tables contenant un champ CLOB ou un BLOB, certaines méthodes DAO générées seront suffixées par «withBLOB » et « withoutBLOB » et utiliseront des vo également suffixés. Le code permet donc d'être optimisé dans le cas où on ne souhaite pas accéder aux BLOBS.

3.5.3 INTEGRATION DANS SPRING

Les classes générées par Ibator utilisent l'API de Spring-DAO (Template de code, injection de la dataSource, …)

Procédure :

- Créer un fichier appContext-dao.xml dans le source-folder (src) et ayant cette structure :

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Configuration de  Spring, spécialement pour les couches DAO :
          - précise la configuration iBatis,
          - précise le mode de connexion utilisé (ici jdbc direct)
          - et indique à Spring d'utiliser le mode "transaction par annotation"

le fichier appContext-dao est référencé par le web.xml
-->
<beans xmlns="http://www.springframework.org/schema/beans"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:tx="http://www.springframework.org/schema/tx"
               xsi:schemaLocation="
                    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                    http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
                    http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd"
    >
```

```
        <!-- ======================== RESOURCE DEFINITIONS ========================= -->

        <!-- Mode NORMAL : DataSource JNDI, à définir comme ressource du serveur JEE -->
        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/ProtoDS"/>

        <!-- Mode TEST UNITAIRE : Datasource locale gérée par Spring, sans pool de connexion -->
        <!--bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
destroy-method="close">
                <property name="driverClassName" value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost:3306/sample" />
                <property name="username" value="sample"/>
                <property name="password" value="sample"/>
        </bean-->

        <!-- SqlMap setup for iBATIS Database Layer -->
        <bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
      <property name="configLocation">
          <value>classpath:ibatis.xml</value>
      </property>
      <property name="useTransactionAwareDataSource">
          <value>true</value>
      </property>
        </bean>

        <bean id="sqlMapClientTemplate"
class="org.springframework.orm.ibatis.SqlMapClientTemplate">
                <property name="sqlMapClient" ref="sqlMapClient" />
                <property name="dataSource" ref="dataSource" />
        </bean>

        <!-- Transaction manager pour une unique DataSource JDBC -->
        <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
                <property name="dataSource" ref="dataSource"/>
        </bean>

        <!--
                Instruct Spring to perform declarative transaction management
                automatically on annotated classes.
        -->
        <tx:annotation-driven transaction-manager="txManager"/>

        <!-- ======================== couche DAO IBATIS ======================== -->
</beans>
```

- Paramétrer la dataSource
- A la fin du fichier, déclarer les beans DAO générés par Ibator et qui seront utilisé par l'application :

```
<bean id="ProductDAO" class="acube.projet.integration.dao.ProductDAOImpl">
        <property name="sqlMapClientTemplate" ref="sqlMapClientTemplate" />
</bean>
```

- Créer un fichier ibatis.xml dans le source-folder (src), afin de déclarer les fichiers sqlMap.xml générés par Ibator:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig
PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<!-- Configuration iBatis, chargé par SpringDAO dans appContext-dao-iBatis.xml -->
<sqlMapConfig>
        <settings
                useStatementNamespaces="true"
        />
        <sqlMap resource="acube/projet/integration/dao/maps/PRODUCT_SqlMap.xml"/>
        <sqlMap
resource="acube/projet/integration/dao/maps/procedures_SqlMap.xml"/>
</sqlMapConfig>
```

- Ne pas oublier de charger appContext-dao.xml à l'initialisation de Spring. Dans une application Web, il faudra l'ajouter au web.xml :

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath*:appContext.xml classpath*:appContext-dao.xml</param-
value>
</context-param>
```

3.5.4 DEVELOPPEMENT SPECIFIQUE IBATIS

Si les DAO, VO et SQLMaps.xml générés par Ibator ne suffisent pas, il est possible d'ajouter du code directement dans les fichiers générés, ou dans des fichiers séparés pour plus de lisibilité.

Règles à respecter :

- Ecrire du SQL standard tant que c'est possible (i.e. : éviter les appels de fonction ou procédures stockées spécifiques au SGBD), afin d'améliorer la portabilité et ainsi permettre l'utilisation de HSQLDB pour les tests unitaires.
- Les requêtes avec jointure doivent utiliser la syntaxe (LEFT OUTER) JOIN … ON et non la forme « FROM A,B »
- **On pourra mapper les jointures sur un graphe d'objet BO uniquement si l'isolation par les VO n'est pas nécessaire (gain de temps).**
- Si une requête utilise du SQL spécifique SGBD, ajouter à son id un suffixe correspondant au nom du sgbd :

```
<insert id="insertProduct-ORACLE" parameterClass="com.domain.Product">
        insert into PRODUCT (PRD_ID,PRD_DESCRIPTION)
                values (#id#,#description#)
        <selectKey resultClass="int" type="pre" >
                SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM DUAL
        </selectKey>
</insert>
```

3.5.4.1 AJOUT D'UNE REQUETE

Règle : Vérifier que la requête ne peut pas être réalisée via une des méthodes générées, notamment **selectByExample, insertByExample, updateByExample, deleteByExample**

3.5.4.1.1 select (avec jointure)

Règle : si une requête de sélection comporte des jointures, elle doit figurer dans le fichier SqlMap.xml correspondant à la table principale de la requête.

Règle : Les données renvoyées par une requête de jointure peuvent être mappées sur un VO spécifique à cette jointure, ou directement un BO.
On pourra soit utiliser les alias « as » SQL pour le mapping des colonnes sur les propriétés de la classe ou définir un ResultMap. L'intérêt du ResultMap est d'être réutilisable et pouvoir gérer les graphes d'objet.

Exemple mapping sur un VO spécifique :

- Ajouter dans le fichier person_SqlMap.xml :

```
<select id="getPersonWithRole" parameterClass=”int” resultClass="acube.projet.integration.vo.PersonWithRole">
SELECT    PER_ID as id,
          PER_FIRST_NAME as firstName,
          PER_LAST_NAME as lastName,
          PER_BIRTH_DATE as birthDate,
          PER_WEIGHT_KG as weightInKilograms,
          PER_HEIGHT_M as heightInMeters,
          ROL_NAME as roleName
FROM PERSON
JOIN ROLE on PERSON.ROL_ID = ROLE.ROL_ID
WHERE PER_ID = #value#
</select>
```

- Créer une classe java **acube.projet.integration.vo.PersonWithRole** avec les propriétés nécessaires.

3.5.4.1.2 insert/update/delete

Règle : Les données à mettre à jour doivent être modélisées par une classe Java, dont le nom est suffisamment explicite. La mise à jour portant sur une table à la fois, on pourra réutiliser la classe VO générée appropriée.

Chaque paramètre de requête #param# correspond au nom d'une propriété Javabean.

Exemple :

```
<insert id="insertPerson" parameterClass="projet.integration.vo.Person">
INSERT INTO
          PERSON (PER_ID, PER_FIRST_NAME, PER_LAST_NAME,
                    PER_BIRTH_DATE, PER_WEIGHT_KG, PER_HEIGHT_M)
          VALUES (#id#, #firstName#, #lastName#,
                    #birthDate#, #weightInKilograms#, #heightInMeters#)
```

```
</insert>

<update id="updatePerson" parameterClass="projet.integration.vo.Person">
UPDATE PERSON
        SET PER_FIRST_NAME = #firstName#,
        PER_LAST_NAME = #lastName#, PER_BIRTH_DATE = #birthDate#,
        PER_WEIGHT_KG = #weightInKilograms#,
        PER_HEIGHT_M = #heightInMeters#
WHERE PER_ID = #id#
</update>

<delete id="deletePersonById" parameterClass="projet.integration.vo.Person">
DELETE PERSON
WHERE PER_ID = #id#
</delete>
```

3.5.4.1.3 Construction dynamique de requête SQL

On pourra s'inspirer du code selectByExample généré par IBatis.

3.5.4.2 APPEL D'UNE PROCEDURE OU FONCTION STOCKEE

Exemples de procedures stockées (MySQL) :

```
DELIMITER $$

CREATE PROCEDURE sp_swap(INOUT a DECIMAL,INOUT b DECIMAL)
BEGIN
        DECLARE t DECIMAL;
        set t = b;
        set b = a;
        set a = t;
END $$

CREATE PROCEDURE sp_productsMaxPrice(price DECIMAL)
BEGIN
        SELECT * from PRODUCT where PRODUCT_PRICE < price;
END $$

CREATE FUNCTION sf_exp(val FLOAT) RETURNS FLOAT
BEGIN
        RETURN 1 + val*(1 + val/2*(1 + val/3*(1 + val/4)));
END $$

CREATE PROCEDURE sp_productsMaxPrice2(price DECIMAL)
BEGIN
        SELECT * from PRODUCT where PRODUCT_PRICE < price;

        return price;
END $$
```

```
DELIMITER ;
```

3.5.4.2.1 déclaration dans SqlMap.xml :

```
<resultMap class="Product" id="ProductResult">
  <result column="PRODUCT_ID" jdbcType="INTEGER" property="productId" />
  <result column="PRODUCT_LABEL" jdbcType="VARCHAR" property="productLabel" />
  <result column="PRODUCT_PRICE" jdbcType="DECIMAL" property="productPrice" />
  <result column="PRODUCT_VALIDITY" jdbcType="TIMESTAMP" property="productValidity" />
</resultMap>

<parameterMap class="java.util.Map" id="map1">
  <parameter property="price" mode="IN"/>
</parameterMap>

<parameterMap class="java.util.Map" id="map2">
  <parameter property="a" mode="INOUT"/>
  <parameter property="b" mode="INOUT"/>
</parameterMap>

<parameterMap class="java.util.Map" id="map4">
  <parameter property="result" mode="OUT"/>
  <parameter property="value" mode="IN"/>
</parameterMap>

<parameterMap class="java.util.Map" id="map5">
  <parameter property="priceOut" mode="OUT"/>
  <parameter property="price" mode="IN"/>
</parameterMap>


  <!--
    Procédure stockée avec paramètre IN retournant un resultSet
  -->
<procedure id="sp_productsMaxPrice" parameterMap="map1" resultMap="ProductResult">
       {call sp_productsMaxPrice(?)}
</procedure>

  <!--
    Procédure stockée effectuant un échange de ses paramètres (INOUT)
  -->
<procedure id="sp_swap" parameterMap="map2">
       {call sp_swap(?,?)}
</procedure>

  <!--
    Fonction stockée effectuant une approximation (grossière) de l'exponentielle
  -->
<procedure id="sf_exp" parameterMap="map4">
       {? = call sf_exp(?)}
</procedure>
```

```
    <!--
      Fonction stockée en utilisant les paramètres nommés
    -->
  <procedure id="sf_exp2" parameterClass="java.util.Map">
         {#result,mode=OUT# = call sf_exp(#value#)}
  </procedure>

    <!--
      Mélange de procédure et de fonction
    -->
  <procedure id="sp_productsMaxPrice2" parameterMap="map5" resultMap="ProductResult">
         {? = call sp_productsMaxPrice2(?)}
  </procedure>

    <!--
      Mélange de procédure et de fonction avec paramètres nommés
    -->
  <procedure id="sf_exp3" parameterClass="java.util.Map" resultMap="ProductResult">
         {#priceOut,mode=OUT# = call sp_productsMaxPrice2(#price#)}
  </procedure>
```

-> Les modes de paramètre possibles sont IN (paramètre d'entrée), OUT (paramètre de sortie) ou INOUT (les deux). Le mode par défaut est IN.

3.5.4.2.2 Classe DAO :

Règles :

- Si la procédure retourne un result-set interne (sp_productsMaxPrice), utiliser les methodes queryForList() ou queryForObject() pour le récupérer
- Si la procédure utilise des paramètres OUT ou INOUT sans rien renvoyer, utiliser la méthode update.
- Dans le cas d'une fonction, déclarer la valeur de retour comme premier paramètre de type OUT et utiliser la méthode update :

```
public class ProceduresDaoImpl extends SqlMapClientDaoSupport implements ProceduresDao {

    /** {@inheritDoc} */
    @SuppressWarnings("unchecked")
    public List<Product> selectProductsMaxPrice(double price) {
        Map<String, Object> maps = new TreeMap<String, Object>();
        maps.put("price", price);
        return getSqlMapClientTemplate().queryForList("procedures.sp_productsMaxPrice",maps);
    }

    public void swap(double[] a, double[] b) {
        Map<String, Object> maps = new TreeMap<String, Object>();
        maps.put("a", a[0]);
        maps.put("b", b[0]);
        getSqlMapClientTemplate().update("procedures.sp_swap",maps);
        a[0] = new Double((String)maps.get("a"));
        b[0] = new Double((String)maps.get("b"));
    }
```

```
    public float exp(float v) {
        Map<String, Object> maps = new TreeMap<String, Object>();
        maps.put("value", v);
        getSqlMapClientTemplate().update("procedures.sf_exp",maps);
        return (Float) new Float((String)maps.get("result"));
    }
}
```

- Dans le cas d'une procédure qui renvoie et un result-set et une valeur de retour et/ou des paramètres OUT ou INOUT, utiliser les méthodes queryForList() ou queryForObject().

Remarques :

Les curseurs peuvent également être mappés sur le type JDBC « OTHER »

3.5.4.3 N+1 SELECTS

Le Problème des N+1 selects apparait lorsqu'on souhaite charger un graphe d'objet. Par exemple quand on charge une liste de N objets avec 1 requête puis pour chaque objet, on effectue une nouvelle requête pour charger un objet lié. Pour résoudre ce problème on utilisera une jointure et un mapping approprié (ResultMap).

On pourra directement réutiliser le graphe d'objet des BO si l'isolation par les VO n'est pas nécessaire.

- Relation 1-1

  Le but est de Mapper sur un objet contenant une référence à un autre objet.

- Relation 1-N ou M-N

  Le but est de Mapper sur un objet contenant une collection.

  On utilise pour cela un ResultMap avec l'attribut groupBy.

```
<resultMap id="categoryResult" class="acube.projet.business.bo.CategoryBO"
groupBy="id">
        <result property="id" column="CAT_ID"/>
        <result property="description" column="CAT_DESCRIPTION"/>
        <result property="productList" resultMap="ProductCategory.productResult"/>
</resultMap>
<resultMap id="productResult" class=" acube.projet.business.bo.ProductBO">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>
<select id="getCategory" parameterClass="int" resultMap="categoryResult">
select C.CAT_ID, C.CAT_DESCRIPTION, P.PRD_ID, P.PRD_DESCRIPTION
from CATEGORY C
left outer join PRODUCT P
on C.CAT_ID = P.PRD_CAT_ID
where CAT_ID = #value#
</select>
```

3.5.4.4 GESTION DES BATCHS UPDATES

3.5.4.4.1 Principes

Les batchs updates sont des batchs JDBC. Le principe de ces batchs est d'éviter d'exécuter unitairement des ordres SQL de création, modification et suppression, surtout s'ils sont nombreux, afin d'optimiser les temps d'exécution. Pour ce faire on distingue les ordres de création, de modification et de suppression que l'on stocke dans trois listes différentes. Puis, on exécute dans un batch JDBC, par groupe de 50, les ordres de création ensuite ceux de modification et enfin ceux de suppression. Ainsi, on réussit à agréger les ordres SQL et à les exécuter par paquets de 50 et non unitairement ce qui permet de multiplier par dix le gain d'exécution.

3.5.4.4.2 Exemple

Pour réaliser les batch updates il est nécessaire d'utiliser l'interface SqlMapClient afin d'extraire les ordres SQL de chaque requête pour les agréger.

3.5.5 CACHE DE REQUETES

3.5.5.1 PRINCIPES

IBatis permet de charger en mémoire des requêtes dans un cache afin d'éviter de relancer celles-ci si ce n'est pas nécessaire. On peut ainsi diminuer le temps d'exécution et optimiser l'accès aux données.

Il préférable d'utiliser ce cache en lecture seules, uniquement sur les données qui sont majoritairement accédées en lecture et dont la fréquence de mise à jour est faible.

Il faut impérativement définir la taille du cache et son time out pour renouveler les données stockées en mémoires. En effet, il est important déterminer quels sont les requêtes à conserver dans le cache pour estimer la taille de celui-ci. Il faut noter qu'une requête qui comporte plusieurs clauses dynamiques différentes engendre autant de requête à stocker en cache qu'il existe de combinaison possible pour ces clauses. Par exemple pour les requêtes de type « ByExample », il y aura autant de requêtes stockées en cache que de combinaison possible entre tous les critères. Ainsi, s'il existe au maximum deux clauses sur la requête, le cache contiendra quatre requêtes, une pour chaque combinaison possible.

Pour optimiser la mémoire du cache plusieurs astuces sont possibles :

- Appliquer des algorithmes supplémentaires pour améliorer la gestion du cache, tel que l'algorithme LRU qui supprimer du cache les requêtes les moins utilisées.
- Modifier le codes des requêtes de type « ByExample » afin que celles-ci ne retourne que les identifiants. Puis, ajouter un algorithme qui parcourt ce résultat et qui effectue une requête sur les clefs primaires. Ce principe est un peu plus couteux à la première exécution mais il permet de stocker moins de requête dans le cache et de stocker tout l'objet.

Enfin, l'utilisation d'un cache n'a pas d'intérêt si celui-ci n'est pas utilisé à plus de 50%.

3.5.6 LIMITER LE NOMBRE DE RESULTAT

3.5.6.1 LIMITER SUR LE SGBD

- Cas oracle :
    - Les 100 premiers résultats d'une requête non triée :

```
SELECT *
FROM `t_client`
WHERE ROWNUM <= 100
```

    - Les 100 premiers résultats d'une requête triée :

```
SELECT *
FROM (SELECT * FROM `t_client` ORDER BY name)
WHERE ROWNUM <= 100
```

Le compteur ROWNUM, initialisé à 1, est définit pour chaque résultat d'une requête. La valeur de celui-ci est attribuée puis incrémenté de 1 à chaque ligne retournée par la requête. Lorsque les données retournées ne sont pas triées avec le mot-clé ORDER BY ce principe ne pose pas de problème. Par contre, si on souhaite trier les données et tenir compte de l'ordre alors il est nécessaire d'utiliser une sous-requête. En effet, la valeur de ROWNUM est attribuée pour chaque ligne **avant** le tri final. Ainsi, une fois le résultat ordonné, le critère des 100 premiers ne peut être respecté.

- Cas ms sql serveur 2005 :
    - Les 100 premiers résultats

```
SELECT TOP 100 *
FROM t_client
```

- Cas mySQL :
    - Les 100 premiers résultats :

```
SELECT *
FROM `ARCHIVA_ARTIFACT`
LIMIT 100
```

3.5.7 PAGINATION DE LISTES

Plusieurs solutions sont possibles pour la pagination. Le choix se fait en fonction des volumes des données et fréquences d'utilisation et de mise à jour.

Les trois variantes les plus communes sont présentées ici.

3.5.7.1 PAGINER SUR LE SGBD

- Cas oracle :

Il est possible d'utiliser la fonction analytique ROW_NUMBER, pour filtrer le résultat d'une requête sur des intervalles consécutifs. Le principe est assez similaire à ROWNUM. En effet, la fonction ROW_NUMBER est appliquée sur le résultat final de la requête alors que ROWNUM est appliqué pendant l'exécution de celle-ci.

Exemple qui affiche 30 lignes à partir de l'enregistrement 10 :
    - avec ROWNUM :

```
SELECT …
FROM (
        SELECT …, ROWNUM num FROM
                (SELECT … FROM t_client ORDER BY name)
        )
WHERE num BETWEEN 10 and 40
```

- avec ROW_NUMBER :

```
SELECT …
FROM (
        SELECT …, ROW_NUMBER() OVER (ORDER BY name) num
        FROM t_client
        )
WHERE num BETWEEN 10 and 40
```

Référence : http://oracle.developpez.com/faq/?page=3-1#rownum

- Cas ms sql serveur 2005 :

Affiche 30 lignes à partir de l'enregistrement 10 :

```
SELECT * FROM (
          SELECT TOP 10 Field1, Field2 FROM (
          SELECT TOP 30 Field1, Field2
          FROM  matable
         ORDER BY monchamp asc
         ) AS tbl1 ORDER BY monchamp desc
         ) AS tbl2 ORDER BY monchamp asc
```

Référence : http://sqlserver.developpez.com/faq/?page=Jeu#Jeu1

Autre approche :

```
select * FROM (
        select *, row_number() over(order by mon_champ) as rownum from maTable
) orderedResults where rownum between 10 and 40
```

- Cas mySQL :

Utilisation de l'instruction **LIMIT** <indice de départ dans la liste de résultat >, <nombre de résultat à afficer >

Exemple, à partir de l'élément 10, afficher 30 résultats :

```
SELECT *
FROM `ARCHIVA_ARTIFACT`
LIMIT 10 , 30
```

avantages	Inconvénient
Ne retourne que le resultset de la page souhaitée	-on exécute la requête à chaque fois (ou du moins à

	chaque changement de page) -Dépendant SGBD

3.5.7.2 PAGINER SUR LE SGBD (2)

Si requête couteuse en temps d'exécution : utiliser une table partagée par les utilisateurs pour stocker le résultat complet de la requête , ensuite la pagination est faite sur cette table selon une technique précédente..

avantages	Inconvénient
on exécute la requête une seule fois	Espace disque plus important, et à priori une table par requête (mais on pourrait ne stocker que les ids du résultat)

3.5.7.3 PAGINER AVEC IBATIS

iBatis propose une fonctionnalité « PaginatedList », qui est à éviter (deprecated dans les versions ultérieures)

Deux méthodes existent selon la variante :

- A - La pagination consiste à afficher les n items d'une requête SQL,

  Cette variante est bien adaptée dans le cas où les données de la liste ne change pas fréquemment.

  Dans ce mode, il est possible d'accéder à une page précise.

- B - La pagination consiste à récupérer les n items suivants le dernier élément de la liste partielle affichée (ou …n items précédents le premier …),

  Cette variante est bien adaptée dans le cas où la fréquence de changement des données de la liste est suffisante importante pour perturber un accès par page.
  Dans ce mode, il n'est pas possible d'accéder à une page précise.

La variante A s'implémente avec la méthode queryForList(stmt, param, Skip, Max) en précisant le skip et max (cf. javadoc), le pseudo d'algorithme d'Ibatis est grosso modo :

- 1. execute Query
- 2. faire « skip » fois next sur le resultset
- 3. mappe les « max» enregistrements suivants

La variante B s'implémente avec la méthode queryWithRowHandler(stmt, param, rowh) (cf. javadoc), le pseudo d'algorithme d'Ibatis est grosso modo :

- 1. execute Query
- 2. Pour chaque ligne, mappe la ligne et appel le rowHandler, qui sélectionne la ligne ou non selon le critère.

Avantages	Inconvénient
Indépendant SGBD Pas de modification de la requête	on exécute la requête à chaque fois On lit tout le resultset pour arriver à la page qui nous intéresse. Pas de modif concurrente

3.5.7.4 PAGINER SUR LE SERVEUR

On récupère le resultset complet de la requête, et on pagine sur le serveur.

Avantages	Inconvénient
on exécute la requête une seule fois Indépendant SGBD	Tout le resultset est en session Pas de modif concurrente

3.5.8 TYPES JAVA SPECIFIQUES

Il est parfois utile d'avoir des types spécifiques dans des BO. Par exemple on peut prendre le cas d'une date stockée en tant que varchar dans la base de données et que l'on souhaite traité de façon particulière en java.

Considérons que nous avons une classe DateIncomplète qui correspond à notre type. Les objets métier de notre application vont contenir des variables de type DateIncomplète.

Pour qu'Ibatis puissent gérer correctement ce type de champs indépendamment de la structure de l'objet DateIncomplete, il va falloir créer un handler particulier pour ce type de champs qui va implémenter l'interface TypeHandlerCallback.

Exemple :

```
public class DateIncompleteTypeHandler implements TypeHandlerCallback {

    /** {@inheritDoc} */
    public Object getResult(
            ResultGetter arg0) throws SQLException {

        String value = arg0.getString();

        DateIncomplete dateIncomplete = new DateIncomplete();
        dateIncomplete.setValue(value);

        return dateIncomplete;
    }

    /** {@inheritDoc} */
    public void setParameter(
            ParameterSetter arg0, Object arg1) throws SQLException {

        if (arg1 == null) {
            arg0.setNull(Types.VARCHAR);
        } else {
            DateIncomplete dateIncomplete = (DateIncomplete) arg1;
            arg0.setString(dateIncomplete.getValue());
        }
    }

    /** {@inheritDoc} */
```

```
    public Object valueOf(
            String arg0) {

        return arg0;
    }

}
```

Le handler comporte trois méthodes :

- getResult : qui permet de transformer les données de la base dans le type Java correspondant (DateIncomplete dans notre exemple
- setParameter : qui permet de définir le paramètre de requête à partir du type Java (varchar (String) dans notre cas)
- valueOf : qui sert à quelque chose mais je ne sais pas trop

Ensuite il suffit dans la configuration de Ibatis de lui préciser que pour tous les champs de type DateIncomplete il faut utiliser le DateIncompleteTypeHandler. Ceci s'effectue dans le ibatis.xml

```
<typeHandler javaType="acube.projet.utility.DateIncomplete"
        callback="acube.projet.integration.dao.utility.DateIncompleteTypeHandler"
/>
```

3.6 SOCLE TECHNIQUE

3.6.1 GESTION DES RESSOURCES

Les ressources peuvent être de deux natures différentes :

- Ressource locale
    - <Resource> enfant de l'élément <Context>
    - name : permet de ranger la ressource dans l'arbre JNDI
    - auth : prend pour valeur « Container » ou « Application » pour désigner qui de Tomcat ou du programmeur doit fournir les accréditations pour accéder à la ressource
    - type : la classe Java qui détermine le reste de la configuration de la ressource

- Ressource globale
    - <Resource> enfant de l'élément <Server>/<GlobalNamingResources>
    - La configuration se fait comme pour une ressource locale
    - Dans le contexte : <ResourceLink>
        - type : reprise de celui de la ressource globale
        - name : nom JNDI accessible par l'application
        - global : référence au nom utilisé dans la partie globale

Les sources de données sont gérées comme des sources locales et sont à définir dans META-INF/context.xml

```
<Context path="/PROJET" reloadable="true">
        ...
        <Resource name="jdbc/demoGlobal" type="javax.sql.DataSource"
        auth="Container" username="root" passeword="root"
        driverClassName="net.sourceforge.jtds.jdbc.Driver"
        url="jdbc:jtds:sqlserver://localhost:1433/DEMO" initialSize="2"
        maxActive="9" maxIdle="4" minIdle="2" maxWait="5000" />
</Context>
```

Le nœud Context contient les caractéristiques suivantes :

- path : discriminant dans l'url qui identifie l'application
- reloadable : scrutation des répertoires classes et lib pour recharger le contexte dynamiquement en cas de modifications.

3.6.2 GESTION DE LA SECURITE

- Utilisation du standard JEE (Tomcat Realm), couplé avec l'intercepteur RolesInterceptor (restriction d'accès aux flux XML, selon profil utilisateur)
- Utilisation de JAAS-Realm
- gestion de Single-Sign-On (SSO) , via des serveurs de type CAS.
- Utilisation de Acegi (Spring Security)

3.6.3 GESTION DES TRANSACTIONS

Configuration Spring src/appContext-dao.xml :

```
        <bean id="sqlMapClient"
class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
        <property name="configLocation">
            <value>classpath:sqlConfig.xml</value>
        </property>
        <property name="useTransactionAwareDataSource">
            <value>true</value>
        </property>
        </bean>
        <!-- Transaction manager pour une unique DataSource JDBC -->
        <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
                <property name="dataSource" ref="dataSource"/>
        </bean>

        <!--
```

```
            Instruct Spring to perform declarative transaction management
            automatically on annotated classes.
    -->
    <tx:annotation-driven transaction-manager="txManager"/>
```

-démarcation des transactions au niveau des services en utilisant les annotations.

-annotation de niveau classe : toutes les méthodes sont transactionnelles

```
@Transactional
// exécution de chaque méthode dans une transaction
public class ProductServiceImpl implements ProductService {
```

-annotation de niveau méthode

```
    @Transactional
    // optimisation : transaction en lecture seule
    public List<Product> list() {
```

Cela permet de surcharger la définition de niveau classe. Notamment pour redéfinir la propagation (création d'une nouvelle transaction, réutilisation, …) ou le niveau d'isolation (READ_UNCOMMITED, …)

3.6.4 GESTION DES ERREURS

3.6.4.1 ERREURS TECHNIQUES

Concerne les erreurs non fonctionnelles, non prévues et souvent irrécupérables du type rupture de communication réseau (Apache, Navigateur Web, Base de de donnée), perte de session, file not found, out of memory, null pointer, et les Runtime Exception en général.

Le framework génère automatiquement un rapport d'erreur détaillé dans la log, et un message simplifié destiné à l'utilisateur incluant un identifiant unique de l'erreur produite (au format « instance de serveur »-« date système »-« numéro aléatoire »).

3.6.4.2 ERREURS FONCTIONNELLES

Les erreurs fonctionnelles concernent le métier de l'application, elles doivent donc remonter à l'utilisateur avec un niveau de détail élevé.

Il est possible de lever une erreur fonctionnelle soit dans la couche Service, soit Action. Pour cela il suffit d'ajouter le code :

throw new BusinessException(« monCodeErreur », new String[]{argument1, argument2, …}) ;

Cette exception remonte la pile d'exécution pour être interceptée par la couche Web, qui se charge de transmettre un message clair pour l'utilisateur (via le result « error »). Le message d'erreur sera recherché dans les fichiers de propriétés (I18N) dans la portée de l'action en cours (cf. section 3.6.5.2)

3.6.5 GESTION DE L'ENCODING ET DE L'INTERNATIONALISATION

3.6.5.1 ENCODING

- IDE (Eclipse, SVN):

Par défaut Eclipse utilise l'encoding de windows, compatible avec ISO-8859-1, pour lire et écrire les fichiers sur disque. Pour utiliser un autre encodage (comme UTF-8, sans BOM) :

clic droit sur le projet / resource / Text file encoding,

Attention : ne modifier les fichiers qu'avec Eclipse ou un éditeur externe reconnaissant l'encodage spécifié (et pas notepad !).

Attention : s'assurer que l'encodage est conservé lors de la sauvegarde de le référentiel SVN.

- Fichiers XML :

  Certains navigateurs utilise l'encoding dans l'entete de fichier XML pour le parsing. Il faut donc le préciser :

```
<?xml version="1.0" encoding="UTF-8"?>
```

- Fichiers XSL et transformation XSLT :

  Il est possible d'indiquer au processeur XSLT l'encodage du flux à générer (mais attention car le processeur peut également le surcharger) :

```
<xsl:output encoding="UTF-8" />
```

- Encoding JAVA

  Les types String et Character (ou char) sont codées en unicode UTF-16 : chaque caractères est stocké sur au moins 2 octets. 4 pour certains caractères.

  Pour ses entrées/sorties, Java utilise l'encodage défini dans la propriété « file.encoding ». Cette dernière reçoit par défaut celle de l'OS au lancement de la JVM.

  Mais il est possible de surcharger l'encodage par défaut au cas par cas.

  Exemple : lecture d'un fichier encodé en UTF-8 :

```
       BufferedReader r = new BufferedReader(new
InputStreamReader(this.getClass().getResourceAsStream("test.txt"),"UTF-8"));

       String s = r.readLine();
```

  Remarque : il est aussi possible de lancer la JVM avec –dfile.encoding=UTF-8, mais toutes les entrées/sorties se feront en UTF-8. Donc si un fichier est ISO, il faudra également surcharger la lecture.

- encoding HTTP coté Serveur :

  Permet de spécifier le décodage des paramètres de requêtes et l'encodage des réponses http.

  Dans struts.xml, ajouter la ligne suivante (Attention : par défaut Struts utilise UTF-8):

```
       <constant name="struts.i18n.encoding" value="ISO8859-15"/>
```

- encoding HTTP coté client, Fichers HTML et Javascript :

  Permet de spécifier l'encodage utilisé par le fichier HTML et les données soumises par les formulaires.

  - Ajouter dans le fichier html , tag <head>:

```
       <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

  - Utiliser le mode POST (sinon encoder les paramètres via encodeURIcomponent)

L'usage des entités html est à éviter, sauf lorsqu'il y a ambiguité (&gt ; & lt ; pour différencier des balises,   ; pour l'espace insécable, …). Si des caractères accentués sont utilisés : paramétrer un encodage ISO ou UTF-8 comme décrit précédemment.

- encoding base de donnée :

  Il n'est pas nécessaire de créer une base en UTF-8 même si l'application est en UTF-8. Ce n'est pas toujours possible (base existante partagée par plusieurs application,…), et cela peut diminuer les performances. Dans ce cas il faudra préciser le charset pour le décodage au niveau du driver, ou le faire manuellement en java…
  Les SGBD possèdent aussi des type compatible unicode (exemple nvarchar pour SQL Server). Il est donc possible d'ajuster finement l'encodage (solution recommandée)

- Encoding driver JDBC :
  On peut spécifier le charset utilisé le décodage des résultats de requêtes SQL si la base et l'application n'utilisent pas le même encoding.

  Par exemple : pour traduire une colonne ISO-8859-1 en chaine java UTF-8, pour mySQL, on modifira l'url de connexion :

```
jdbc:mysql://localhost:3306/sample?characterEncoding=UTF-8
```

Remarques :

UTF-8 permet d'optimiser les flux contenant beaucoup de caractères Latin, contrairement à UTF-16 ou UTF-32. car UTF-8 code les caractères entre 1 et 4 octets (cf. Wikipédia : http://fr.wikipedia.org/wiki/UTF-8). Ainsi tous les caractères US-ASCII se retrouvent dans UTF-8 (car codage sur 1 octet). De même pour tous les caractères de code < 127 de l'ISO-8859-1(5). Par contre les caractères accentués sont codés sur 2 octets. Et le symbole € sur 3 octets.
le caractère « € » n'est géré qu'avec ISO-8859-15 et UTF-8 Attention : les caractères « ÿ » ou « Œ » ne sont pas géré par ISO !!

3.6.5.2 INTERNATIONALISATION (I18N)

- Couche Web :

Les messages (erreurs, information, etc…) sont stockés dans des fichiers de propriétés I18N appelés resource-bundle (RS).

Algorithme de recherche du RS de Struts :

Recherche des RS dans l'ordre suivant :

- le RS associé à l'action : situé dans le même package que la classe action et nommé MonAction_fr.properties
- le RS associé à l'une des classes action parente
- le RS associé au package : package_fr.properties
- le RS associé à l'un des package ancêtre.
- le RS globale de l'application

Si plusieurs RS sont présents, utilisation du RS le plus proche de celui de la localisation de l'utilisateur (Pays + langue).

Utilisation :

- Dans une classe Action, il suffit d'appeler la méthode getText (« mon.code.message »), qui utilise la localisation par défaut de l'utilisateur (envoyée par la requête http).
- Dans un template JSP, on utilise le tag Struts :

```
        <s:property value="text('mon.code.message')" />
```

- Couche Service

Déclaration du préfix des fichiers de propriétés contenant les messages dans le appContext.xml :

```
    <bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="messages"/>
    </bean>
```

Tous les messages seront donc stockés dans des fichiers messages_fr.properties, messages.properties, … .

Ensuite dans le service il suffit de récupérer les messages à partir du context Spring comme ceci :

```
applicationContext.getMessage(
            "Code de la propriété", new Object[]{"parametre1"}, Locale.FRENCH);
```

Pour récupérer l'application Context il faut utiliser l'injection Spring (Interface ApplicationContextAware).

3.6.6 Gestion des Logs

- **Utilisation de Log4J. Le code peut référencer directement l'API Log4J** (notamment le Logger)

- Paramétrage dans /src/log4j.properties ou /src/log4j.xml .

3.7 Gestion des Tests Unitaires

3.7.1 Principes Generaux :

- Créer un source-folder « tst » à la racine du projet.
- Créer une classe de test par classe Action ou Service testée. La classe est suffixée par Test.
- Créer une méthode par scénario de test envisagé (toujours prévoir au moins un cas normal et un cas d'erreur)
- En cas de correction d'anomalie, toujours créer un test unitaire reproduisant l'anomalie avant de la corriger.

3.7.2 Test de classe Action

```
public class ListTest extends BaseStrutsTestCase<List> {

    @Before public void initDB() {
        DataSource ds = (DataSource) applicationContext.getBean("dataSource");
        new JdbcTemplate(ds).execute("DELETE FROM PRODUCT");
    }
```

```
    @Test  public void testScenario() throws Exception {

        tstInitAction( "/flux/protected/products", "add");
        tstRequestParameter(
            "product.productLabel", "aaa");
        tstRequestParameter(
            "product.productPrice", "30.55");
        tstRequestParameter(
            "product.productValidity", "20/01/2010");
        tstExecute();
        Assert.assertEquals(Action.SUCCESS, tstGetResult());

        tstInitAction("/flux/protected/products", "list");
        tstExecute();
        Assert.assertEquals(Action.SUCCESS, tstGetResult());
        Assert.assertEquals(1, tstGetAction().getList().size());

        Product p = tstGetAction().getList().get(tstGetAction().getList().size()-
1);
        tstInitAction("/flux/protected/products", "remove");
        tstRequestParameter(
            "product.productId", ""+p.getProductId());
        tstExecute();
        Assert.assertEquals(Action.SUCCESS, tstGetResult());
    }

    @Test  public void testErreurs() throws Exception {
        tstInitAction("/flux/protected/products", "remove");
        tstRequestParameter("product.productId", "-1");
        tstExecute();
        Assert.assertEquals("errorReport", tstGetResult());//se traduit par une
erreur technique


        tstInitAction("/flux/protected/products", "add");
        tstRequestParameter(
            "product.productLabel", "aaa");
        tstExecute();
        Assert.assertEquals(Action.INPUT, tstGetResult());
        Assert.assertEquals(2,
            tstGetAction().getFieldErrors().size());

        tstInitAction("/flux/protected/products", "add");
        tstRequestParameter(
            "product.productLabel", "aaa");
```

```
        tstRequestParameter(
            "product.productPrice", "30.55");
        tstRequestParameter(
            "product.productValidity", "20/01/2000");
        tstExecute();
        Assert.assertEquals(Action.INPUT, tstGetResult());
        Assert.assertEquals(1,
            tstGetAction().getFieldErrors().size());

    }

}
```

3.7.3 TEST DE SERVICE

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/appContext.xml","/appContext-dao.xml"})
public final class ProductServiceTest {

    /** auto-injection par Spring */
    @Autowired
    private ProductService service;

    @org.junit.Test
    public void totoList() throws Exception {
        List<Product> utils = service.list();
        Assert.assertEquals(1000, utils.size());
    }
}
```

3.8 REPORTING & EDITIQUE

3.8.1 JASPERREPORT

Jasperreport est un framework permettant de générer des rapports sous différents formats:

- PDF
- HTML
- XLS
- RTF (et non DOC)
- CSV
- XML
- TXT

Depuis le LISE 3.1.0, les bibliothèques jasperreport sont incluses en version : 3.5.3.

3.8.1.1 CYCLE DE VIE D'UN RAPPORT

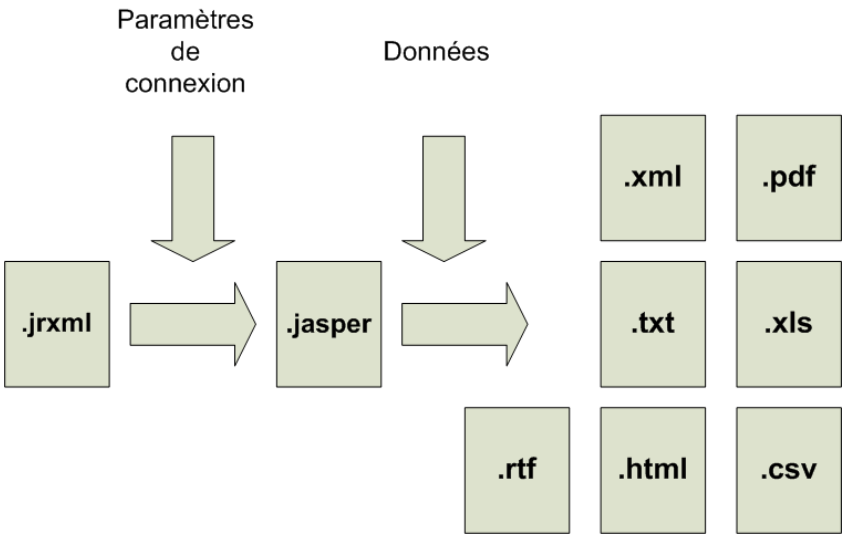


Figure 1 : Cycle de vie d'un rapport Jasper

3.8.1.2 ARCHITECTURE

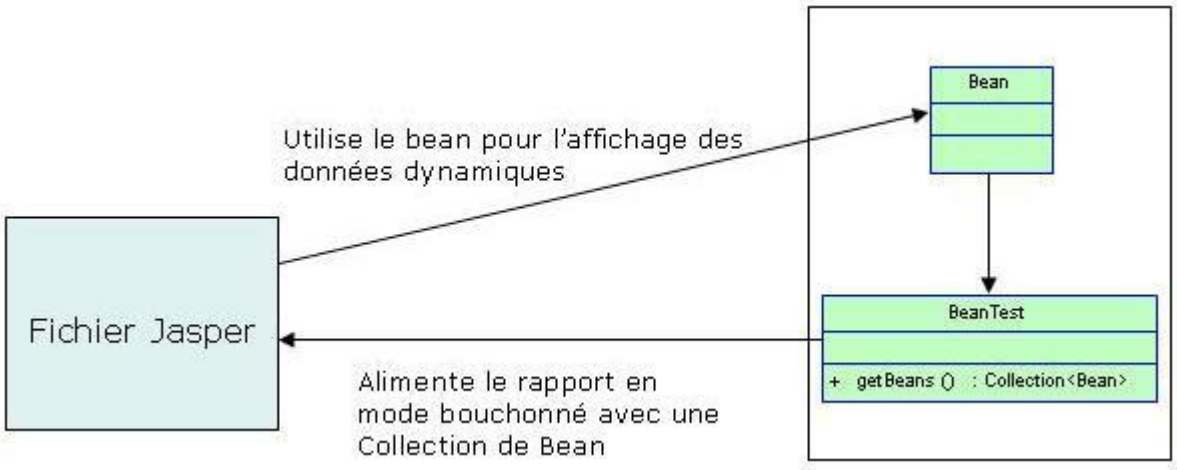


Figure 2 : Bloc création rapport Jasper

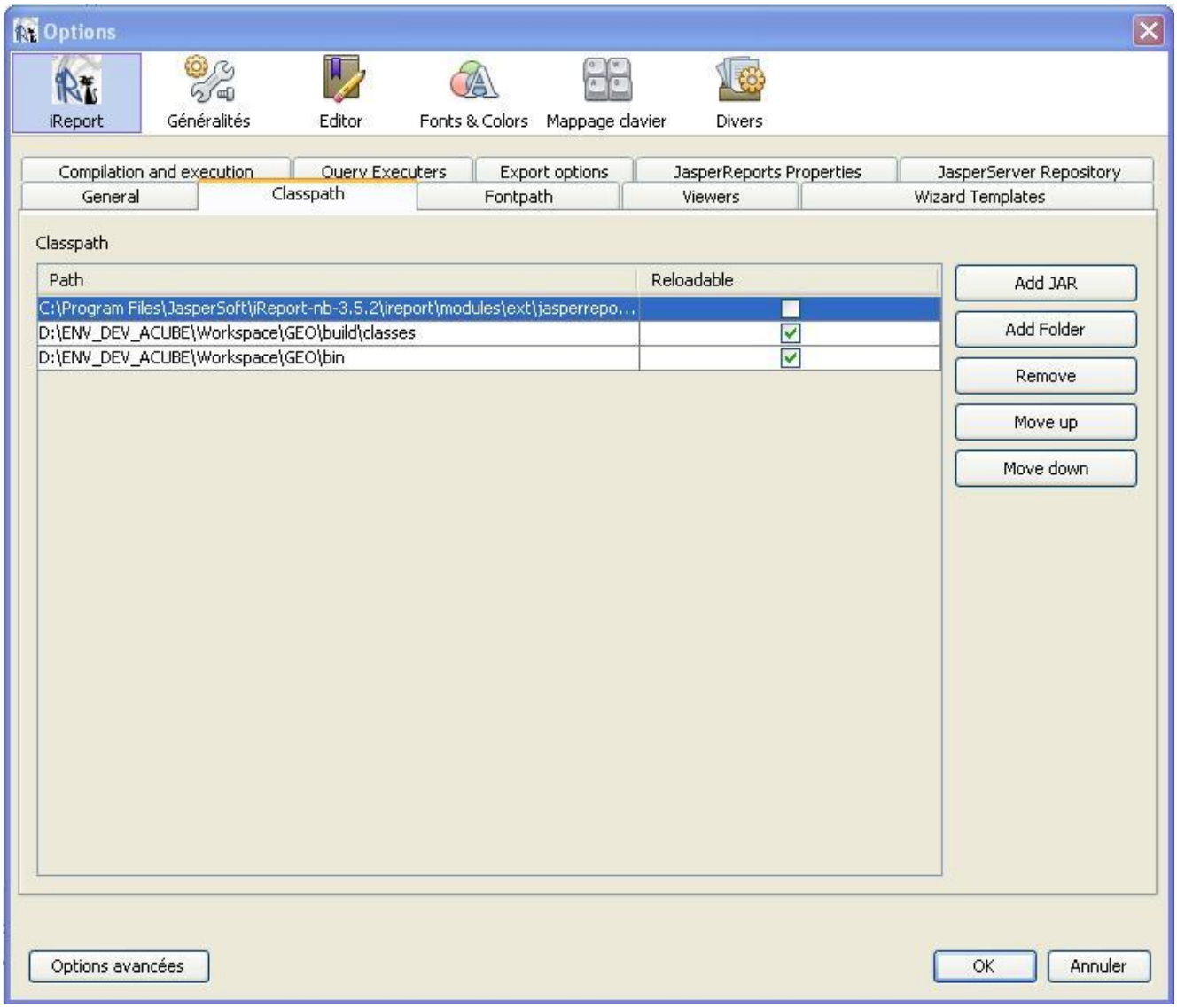
3.8.1.3 CREATION D'UN RAPPORT AVEC IREPORTS

IReport est un outil wysiwing permettant de créer des fichiers (.jrxml) utilisables ensuite par la librairiejasperreport.

3.8.1.3.1 Paramétrage

3.8.1.3.1.1 Classpath

Aller dans le menu Outils/Options, puis allez dans l'onglet Classpath :



Il faut ajouter deux classpath, le premier servira pour pointer sur les Beans, et le second pour pouvoir accéder aux BeanTest afin de pouvoir tester le document en mode bouchonné :

- WORKSPACE_ECLIPSE\Projet\build\classes
- WORKSPACE_ECLIPSE\Projet\bin

3.8.1.3.2 Création des Beans

Avant de commencer à créer le rapport, il faut tout d'abord créer le Bean principal qui alimentera le rapport.
Pour chaque fichier .jasper, il devra y correspondre un BeanTest et Bean.
Dans le cas d'un rapport simple (pas de sous-rapport), un seul Bean suffira. En revanche, si le rapport à créer est plus complexe et nécessite un découpage et une utilisation de sous-rapport, alors le Bean principal contiendra autant de List de « BeanSousRapport » qu'il y a de sous-rapport dans le fichier à générer.

3.8.1.3.2.1 Bean

Le Bean est en quelque une image Objet du rapport que l'on devra créer. Les variables définies dans cet objet seront utilisées au sein du rapport.

3.8.1.3.2.2 Bean Test

Le BeanTest est une classe contenant une méthode statique retournant une Collection de Bean. Cette classe permet de tester le rapport en mode bouchonné.

```
public class LigneElectionBOTest {

    public static List<LigneElectionBean> getBeans() {

        List<LigneElectionBean> beans = new ArrayList<LigneElectionBean>();

        LigneElectionBean ligneElectionBO;
        for (int i = 0; i < 100; i++) {
            ligneElectionBO = new LigneElectionBean();
            ligneElectionBO.setSigleOrgane("Sigle Organe");
            ligneElectionBO.setLibelleElection("Libellé Election");
            ligneElectionBO.setNomCandidat("Nom Candidat");
            ligneElectionBO.setEtatElection("Elu");
            ligneElectionBO.setDateElection(new Date());

            beans.add(ligneElectionBO);
        }
        return beans;
    }
}
```

3.8.2PDF/FDF

Description : Forms Data Format ou FDF est un format de fichier développé par Adobe basé sur le format PDF et qui permet l'insertion de champs de saisie dans un fichier PDF. Ces champs peuvent être valorisés par un humain ou par un programme afin de produire un rapport final.

Intérêt : plus rapide à mettre en œuvre que Jasper ou XSLT pour générer des PDF à partir de formulaires préexistant ayant une complexité élevée (type Cerfa).

Limites :

- Les tableaux dynamiques ne sont pas gérés.
- Il n'est pas possible de masquer ou d'insérer des parties du document (concaténation à gérer à part).

Prérequis : Avoir généré un fichier PDF/FDF avec un outil Adobe ou avec OpenOffice.

Guide Pour OpenOffice : http://documentation.openoffice.org/manuals/userguide3/0215WG3-UsingFormsInWriter.pdf. (Il suffit ensuite d'exporter au format PDF)

Mise en œuvre dans la couche Web :

1) Créer une action Struts pour la génération, qui implémente l'interface FDFAware :

- Méthode **getFdfBean**() : renvoie un bean java ou une Map servant à valoriser les champs du formulaire FDF. Le bean peut très bien être l'action elle-même (return this).
  Le nom d'un champ de formulaire doit correspondre à un attribut du bean ou a une clé de la Map.
  Le nom d'un champ peut être « composé » avec le séparateur « : », par exemple « personne:nom ». Dans ce cas le bean devra contenir une propriété « personne » contenant une propriété « nom », accessibles via les « getter » associés.
  Tous les types java peuvent être utilisés. La conversion en chaine de caractère lors de la valorisation utilise la méthode toString() de l'objet (donc généralement la Locale et un format par défaut).
  Par défaut le type Boolean est converti en « Yes » ou « No » pour pouvoir fonctionner avec la case à cocher.
  Par défaut le type Date est formaté en « dd/MM/yy ».

- Méthode **setUnmergedFields**() : appelée en fin de génération, pour indiquer à l'action la liste des champs non fusionnés.
  Remarque : les champs non fusionnés sont aussi logués avec un niveau « WARN »

2) Dans struts.xml, pour cette action, utiliser le result de type « **fdf** ».

   - Renseigner le chemin du modèle PDF/FDF à utiliser dans l'attribut « **location** », ou directement dans le nœud.

   - Si le chemin commence par « **classpath:**», le modèle doit être situé dans le classpath de l'application avec comme emplacement relatif l'action en cours, sinon le modèle doit être situé dans « WebContent » ou un sous-répertoire :

```
<action name="cerfa"
    class="acube.projet.web.action.export.CerfaAction">
    <result type="fdf">classpath:cerfa-cp.pdf</result>
</action>
```

ou

```
<action name="cerfa"
    class="acube.projet.web.action.export.CerfaAction">
    <result type="fdf">/templates/export/fdf/cerfa-cp.pdf</result>
</action>
```

Remarque : Il est aussi possible d'utiliser la value-stack et les variables struts.

Mise en œuvre classique :

1) Appel à la fonction FDF.export(in, out, data)
   - in est le flux du modèle PDF/FDF
   - out est le flux de sortie
   - data est un javabean ou une Map

Fusion des images :

Pour insérer dynamiquement une image dans le PDF final :

1) créer un champ quelconque dans le formulaire PDF/FDF pour définir la position et la taille de l'image. L'image aura par défaut la taille du rectangle.
2) Dans l'objet de données, créer une propriété liée à ce champ ayant le type FDFImage (classe framework). Un objet FDFImage prend en entrée l'URL de l'image (qui peut être une url http, filesystem ou une ressource JAVA situé dans le classpath), ou directement un tableau de byte (cas du stockage en base par exemple).

   Les formats d'image reconnus sont : GIF, PNG, Bmp, Tiff

3.8.3XSLT

[Reprise de la solution LISE V2]

3.8.3.1 WORD/EXCEL

1) Création d'un modèle de document (au format html compatible office 2000/2003/2007, ou le nouveau format xml office) dans une feuille XSLT :
2) On commence par créer le document dans Office ou OpenOffice, puis on exporte au format html/xml
3) On transforme ce document en feuille XSLT (on isole les parties structurantes dans des templates xslt).
4) On modifie la feuille de manière à ajouter les parties dynamiques (qui seront valorisées à partir des données de l'action Struts)
5) Dans struts.xml, pour l'action qui génère le document, on utilisera result de type « **xslt** ».

3.8.3.1 PDF

1) Création d'un modèle de document XSL/FO « from scratch ».

2) Prévoir les parties dynamiques pendant la construction du modèle (qui seront valorisées à partir des données de l'action Struts)

3) Dans struts.xml, pour l'action qui génère le document, on utilisera result de type « **xsltFop** ».

3.8.4CSV

Utilisation : uniquement pour des données tabulaires brutes (sans style).

3) Création une action Struts pour la génération qui implémente l'interface CSVAware

4) Dans struts.xml, pour cette action, utiliser le result de type « **csv** ».

3.8.5 TEMPLATE DE MAIL

Description :

Le Framework propose une solution de template pour les mails basée sur Vélocity et Spring.

Le moteur de Template Velocity récupère le fichier VTL grâce à un « resource loader » et fusionne le fichier VTL avec les données métiers (objets JAVA).

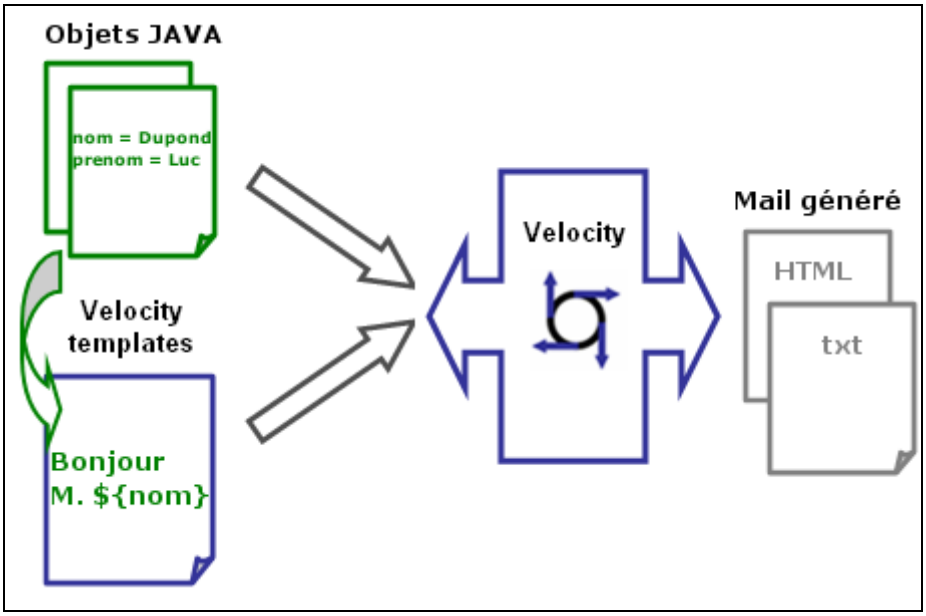


Schéma de fonctionnement de Velocity

Pré requis :

Les bibliothèques nécessaires sont incluses dans le Framework LISE V3 à partir de la version 3.2.0.

Mise en œuvre :

1) Créer un fichier nommé « spring-appcontext-mail.xml » avec au minimum le contenu suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Configuration de Spring Mail -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
            http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
            http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
            http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

<!-- Connexion à la ressource JNDI -->
<bean id="mailSession" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName"><value>java:comp/env/mail/Session</value></property>
</bean>

<!-- Déclaration du bean JavaMail qui envoie les mails  -->
<bean id="mailSender"
class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="session"><ref bean="mailSession"/></property>
```

```
</bean>

<!-- Instanciation du moteur de template avec un ressource loader de type
classpath  -->
<bean id="velocityEngine"
    class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
    <property name="velocityProperties">
        <props>
            <prop key="resource.loader">class</prop>
            <prop key="class.resource.loader.class">
            org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader
            </prop>
        </props>
    </property>
</bean>

<!-- Instanciation du service applicatif contenant l'algorithme d'envoi des mails
-->
<bean id="mailService" class="acube.projet.business.service.MailServiceImpl">
    <property name="mailSender" ref="mailSender" />
    <property name="velocityEngine" ref="velocityEngine" />
</bean>

</beans>
```

Avec cette configuration ci-dessus, le projet disposera d'un service « MailService » contenant :

- un moteur de Template Velocity instancié avec un chargeur de ressources capable de récupérer les fichiers VTL dans le classpath JAVA.
- un objet MailSender, contenant les classes JavaMail instanciées avec les paramètres JNDI (contenu dans le fichier context.xml).

2) Créer le service JAVA qui doit envoyer des mails.

a) Créer l'interface correspondant à la configuration Spring avec une méthode métier « sendAlerteMail » :

```
public interface MailService {

    /**
     * @param velocityEngine
     *            the velocityEngine to set
     */
    public abstract void setVelocityEngine(
            VelocityEngine velocityEngine);

    /**
     * @param mailSender
     *            the mailSender to set
     */
    public abstract void setMailSender(
            MailSender mailSender);

    /**
     * Envoi du mail
     */
    public abstract void sendAlerteMail();
```

```
}
```

Pour l'implémentation, la seule méthode a exécuté pour fusionner la template VTL avec les données métiers est :

```
String VelocityEngineUtils.mergeTemplateIntoString (VelocityEngine, String VTL,Map<String,Object> model);
```

Cette méthode prend trois paramètres :

- Le moteur de template instancié par Spring,
- La clé permettant d'identifier le fichier VTL à récupérer,
- Une Map<String,Object> contenant les objets JAVA métier

b) Créer le service implémentant de l'interface ci-dessus :

```
public class MailServiceImpl implements MailService {

    /**
     * Template engine
     */
    private VelocityEngine velocityEngine;

    /**
     * mailSender
     */
    private MailSender mailSender;

    /**
     *
     */
    public MailServiceImpl() {

    }

    /**
     * @param velocityEngine
     *            VelocityEngine
     * @param mailSender
     *            MailSender
     */
    public MailServiceImpl(
            VelocityEngine velocityEngine, MailSender mailSender) {

        this.velocityEngine = velocityEngine;
        this.mailSender = mailSender;
    }

    /**
     * @param velocityEngine
     *            the velocityEngine to set
     */
    public void setVelocityEngine(
            VelocityEngine velocityEngine) {

        this.velocityEngine = velocityEngine;
    }

    /**
```

```
     * @param mailSender
     *            the mailSender to set
     */
    public void setMailSender(
            MailSender mailSender) {

        this.mailSender = mailSender;
    }



    /**
     * Envoi du mail
     */
    public void sendAlerteMail() {

        //Model
        Map<String, Object> model = new HashMap<String, Object>();
        model.put("user", "Mon Utilisateur");
        model.put("destinataire", "M. Destin");

        //Transform
        String text = VelocityEngineUtils.mergeTemplateIntoString(
            this.velocityEngine, "mailTemplate.vm", model);

        //Send mail
        SimpleMailMessage msg = new SimpleMailMessage();
        String tos[] =
                {"M. Destin<m.destin@maee.com>",
                        "D2 <d2@maee.com>"};

        msg.setTo(tos);
        msg.setFrom("Expediteur" + "<Expedit@acube.com>");
        msg.setText(text);

        this.mailSender.send(msg);

    }

}
```

3) Créer le fichier Template « `mailTemplate.vm` » à la racine des sources avec le contenu suivant :

```
<html>
<body>
<h3>Bonjour ${user.userName},et bienvenue sur le site ${site}!</h3>
<div>
   L'application utilisera désormais l'adresse mail suivantes<a
href="mailto:${user.emailAddress}">${user.emailAddress}</a>.
</div>
</body>
</html>
```

Pour en savoir plus sur la syntaxe, consulter le guide d'utilisation disponible en français sur le site de Velocity : http://velocity.apache.org/engine/releases/velocity-1.6.2/translations/user-guide_fr.html

3.9 AJOUT D'UN CAPTCHA AU PROJET

3.9.1 DESCRIPTION D'UN CAPTCHA

Un Captcha (**J**ava **C**ompletely **A**utomated **P**ublic **T**est to tell **C**omputers and **H**umans **A** part) est un test permettant de différencier de manière automatisée un humain d'un ordinateur, lors du remplissage de données de formulaire d'inscription sur des sites par exemple.

C'est un moyen technique rependu pour lutter contre des robots malveillants.

3.9.2 MODIFICATIONS COTE SERVEUR

Dans le fichier « web.xml », ajouter le fichier de configuration Spring du captcha:

```
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            …
            classpath*:spring-captcha.xml
        </param-value>
    </context-param>
```

Le fichier « spring-captcha.xml » fait appel à la configuration par défaut du captcha du Framework.

Créer une **classe Action** de vérification du Captcha :

```
public class Captcha extends ActionSupport implements ServletRequestAware {

    /**HttpServletRequest*/
    private HttpServletRequest servletRequest;

    /**Booléen indiquant si le captcha est correct*/
    private Boolean validCaptcha;

    /*** Valeur à tester, entrée par l'utilisateur*/
    private String captchaTest;

    /**Captcha Service*/
    private transient DefaultManageableImageCaptchaService captchaService;

    public String execute() {

        // Récupération de l'id du captcha
        String captchaId = this.servletRequest.getSession().getId();

        // On teste si l'utilisateur a entré le bon captcha
        this.validCaptcha = this.captchaService.validateResponseForID(
            captchaId, this.captchaTest);

        return Action.SUCCESS;
    }

      …
}
```

Cette action compare la valeur entrée par l'utilisateur suite à l'affichage du captcha à la valeur réelle du captcha. Le nom de l'action doit être différent de « captcha » car ce nom est utilisé par l'action de génération d'image du Framework.

Struts.xml :

Il faut ensuite ajouter dans le fichier Struts l'appel à l'action de vérification du captcha.

```
    <package name="captchaCheck"
        extends="struts-acube-fwacubej2ee" namespace="/flux/protected/captcha">
         <action name="captchaCheck"
                 class="acube.projet.web.action.captcha.Captcha">
              <result name="success" type="dispatcher">
                  <param name="location">
                    /templates/captcha/captchaCheck.jsp
                  </param>
              </result>
          </action>
    </package>
```

Flux de retour :

Ajouter le flux de retour « captchaCheck.jsp » mentionné dans la configuration Struts ci-dessus.

```
<?xml version="1.0" encoding="UTF-8"?>
<%@ page contentType="text/xml; charset=utf-8" %>
<%@ taglib uri="/struts-tags" prefix="s"%>
<CONTROLE>
    <RESULTAT>
        <VALUE><s:property value="validCaptcha" /></VALUE>
    </RESULTAT>
</CONTROLE>
```

Pour vérifier que l'action de génération d'images du Framework fonctionne correctement, le flux à appeler côté client pour la génération de l'image est :

```
http://{Serveur}/{Projet}/flux/protected/framework/captcha.xml
```

Une image doit ainsi apparaître :

Figure 3 - Exemple d'image de Captcha

Côté client, il faut ajouter les fichiers JavaScript et html, pour l'affichage du formulaire Captcha et l'envoi de la valeur entrée par l'utilisateur (aller voir dans la maquette client riche le composant Captcha)

3.9.3 PERSONNALISER SON CAPTCHA

Pour personnaliser son Captcha, il suffit de créer un fichier Spring de configuration du Captcha dans votre projet, avec un autre nom que « spring-captcha.xml ».

Fichier « spring-captcha-surcharge.xml » :

```
<bean id="captchaService"
      class="com.octo.captcha.service.image.DefaultManageableImageCaptchaService"
      scope="singleton">
      <property name="captchaEngine" ref="captchaEngine" />
</bean>


<!-- Captcha factory : Définition dy type de Factory à utiliser -->
<bean id="captchaEngine" class="com.octo.captcha.engine.GenericCaptchaEngine">
      <constructor-arg index="0">
            <list><ref bean="CaptchaFactory" /></list>
      </constructor-arg>
</bean>

<bean id="CaptchaFactory" class="com.octo.captcha.image.gimpy.GimpyFactory">
      <constructor-arg><ref bean="wordgen" /></constructor-arg>
      <constructor-arg><ref bean="wordtoimage" /></constructor-arg>
</bean>

<!–
Generateur de mots
Il est ainsi possible d’utiliser un dictionnaire ou une suite de lettres
aléatoires
 -->
<bean id="wordgen"
      class="com.octo.captcha.component.word.wordgenerator.DictionaryWordGenerator
">
      <constructor-arg><ref bean="filedict" /></constructor-arg>
</bean>

<!-- Dictionnaire utilisé -->
<bean id="filedict" class="com.octo.captcha.component.word.FileDictionary">
      <constructor-arg index="0"><value>toddlist</value></constructor-arg>
</bean>

<!--
      Utilisation de font generator, background generator, Text paster
-->
<bean id="wordtoimage"
      class="com.octo.captcha.component.image.wordtoimage.ComposedWordToImage">
      <constructor-arg index="0"><ref bean="fontGenRandom" /></constructor-arg>
      <constructor-arg index="1"><ref bean="backGenUni" /></constructor-arg>
      <constructor-arg index="2"><ref bean="simpleWhitePaster" /></constructor-
arg>
</bean>

<bean id="fontGenRandom"
      class="com.octo.captcha.component.image.fontgenerator.RandomFontGenerator">
      <constructor-arg index="0"><value>40</value></constructor-arg>
      <constructor-arg index="1"><value>50</value></constructor-arg>
      <constructor-arg index="2"><list><ref bean="fontArial"
/></list></constructor-arg>
</bean>

<bean id="fontArial" class="java.awt.Font">
      <constructor-arg index="0"><value>Arial</value></constructor-arg>
      <constructor-arg index="1"><value>0</value></constructor-arg>
      <constructor-arg index="2"><value>10</value></constructor-arg>
```

```
</bean>

<bean id="backGenUni"
      class="com.octo.captcha.component.image.backgroundgenerator.UniColorBackgrou
ndGenerator">
      <constructor-arg index="0">
            <value>300</value>
      </constructor-arg>
      <constructor-arg index="1">
            <value>100</value>
      </constructor-arg>
</bean>

<bean id="simpleWhitePaster"
class="com.octo.captcha.component.image.textpaster.SimpleTextPaster" >
    <constructor-arg type="java.lang.Integer"
index="0"><value>3</value></constructor-arg>
    <constructor-arg type="java.lang.Integer"
index="1"><value>5</value></constructor-arg>
    <constructor-arg type="java.awt.Color" index="2"><ref
bean="colorGreen"/></constructor-arg>
</bean>

<bean id="colorGreen" class="java.awt.Color" >
    <constructor-arg index="0"><value>0</value></constructor-arg>
    <constructor-arg index="1"><value>255</value></constructor-arg>
    <constructor-arg index="2"><value>0</value></constructor-arg>
</bean>
```

Chargez ensuite ce fichier, en ajoutant dans le « web.xml » :

```
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            …
            classpath*:spring-captcha-surcharge.xml
        </param-value>
    </context-param>
```

L'ensemble de la configuration du Captcha se fait dans ce fichier (couleur, taille, distorsion, utilisation d'un dictionnaire, langue, etc.…).

Plus d'informations sur le site :
http://forge.octo.com/jcaptcha/confluence/display/general/JCaptcha+and+the+SpringFramework

3.10 PERFORMANCES ET BONNES PRATIQUES

3.10.1 GENERAL :

- **Ne jamais oublier le premier principe, aphorisme de Donald Knuth : « Premature Optimization is the root of all evil ».**

- **Ne pratiquer à priori que les optimisations peu couteuses en développement et les bonnes pratiques suivantes (qui deviendront des réflexes)**

- Ne pas utiliser les classes Vector et Hashtable qui utilisent de manière cachée la synchronisation Java. Utiliser ArrayList, HashMap, TreeMap, HashSet, … Si la collection DOIT être partagée entre plusieurs Threads, utiliser Collection.synchronizedCollection(…)
- Sauf si vous savez exactement à quoi ca sert, n'utilisez pas le mot clé « synchronized »
- Seuls les calculs de montant doivent se faire en BigDecimal (et non Double ou Float pour éviter les erreurs d'arrondi). Pour les identifiants de base de données, le type Long suffit.
- Tirer partie du « Just In Time compiler » de la JVM. Le JIT transforme le code java appelé fréquemment en code natif.

```
public void maFonction() {
    for(int i=0 ; i<100000 ; i++) {
        //traitement
    }
}
```

en

```
public void maFonction() {
    for(int i=0 ; i<100000 ; i++) {
        maFonctionUnitaire(i) ;
    }
}

public void maFonctionUnitaire (int i) {
        //traitement
    }
```

3.10.2 COUCHE WEB :

- Utiliser les JSP au lieu des templates XSL (conso mémoire et temps de réponse plus faible), surtout si les données à renvoyées au client sont importantes.
- Toute requête http de mise à jour devrait également retourner les données permettant de rafraichir le formulaire (plutôt que de renvoyer le flux « success »). Cela évite au client de devoir appeler d'autres flux pour se rafraichir. Pour ce faire, coté serveur, on devra utiliser les techniques de réutilisation d'action et de JSP (chainage d'action, inclusion de jsp).
- Limiter le flux xml d'un tableau à 500Ko non compressé. Penser à définir avec la MOA une limite du nombre de lignes retournée par requête (<1000). Cf. Pagination.

3.10.3 COUCHE SERVICE :

- La signature (arguments et type de retour) doit être la plus abstraite possible. En particulier, les collections (ArrayList, HashMap) devront êtres passée sont forme de type abstrait java.util.Collection, voire java.util.List ou java.util.Map, et non en type concret.

3.10.4 COUCHE DAO :

- Cibler les **requêtes optimisées sur les fonctionnalités les plus utilisées**… Le N+1 select sera souvent acceptable, et bien moins couteux à développer.
- On peut récupérer les associations 1-1 et 1-N par une seule requête avec jointure et mapper le résultat sur un BO (via plusieurs resultMap ibatis).
- Ne rapporter qu'une collection rattachée à un objet par requête (left join). Si un objet possède plusieurs collections, il est plus optimal de récupérer chaque collection séparément.
- Pour les traitements par lots, on pensera à utiliser la pagination lors de la lecture des objets à traiter pour optimiser la mémoire, et les Batchs JDBC (via IBatis) pour optimiser les écritures.

3.10.5 BASE DE DONNEE :

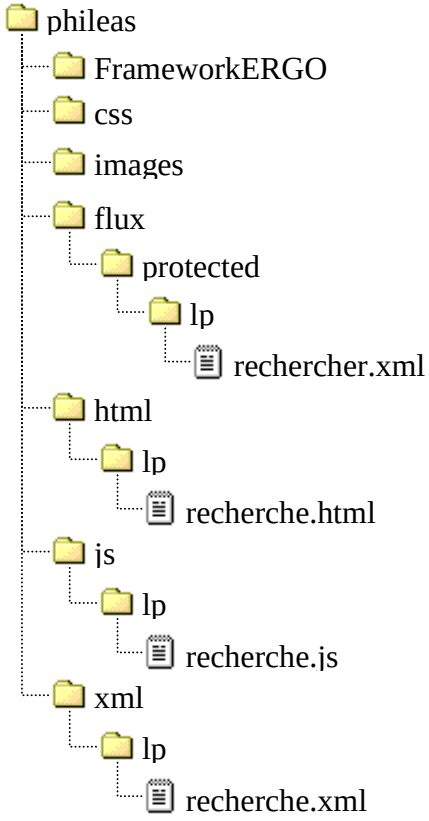
- Pas de procédure stockée, sauf si problème de perf en exploitation qui impose d'en créer.

4 NOMENCLATURE

4.1 CLIENT RICHE

Elément	Nomenclature	Exemple
FICHIER HTML	/html/<bigramme du cas d'utilisation>/<nom du cas d'utilisation>.html	/html/lp/recherche.html
Fichier Javascript	/jsclient/<bigramme du cas d'utilisation>/<nom du cas d'utilisation>.js	/jsclient/lp/recherche.js
Fichier XML statique	/xml/<bigramme du cas d'utilisation>/<nom du cas d'utilisation>.xml	/xml/lp/recherche.xml
Fichier XML dynamique (fichier bouchon)	/flux/protected/<bigramme du cas d'utilisation>/<nom de l'action>.xml	/flux/protected/lp/rechercher.xml

Exemple :



4.2 SERVEUR

La structuration des packages et répertoires se fait par couche applicative d'abord et (éventuellement) par cas d'utilisation ensuite.

Elément	**Nomenclature**	**Exemple**	**répertoire**
Action	acube.projet.web.action.<cas d'utilisation>.nomAction	acube.projet.web.action.gestiondossier.ListerAction	src
Service (interface)	acube.projet.business.service.<cas d'utilisation>.nomService	acube.projet.business.service.DossierService	src
Service (implémentation)	acube.projet.business.service.<cas d'utilisation>.nomServiceImpl	acube.projet.business.service.DossierServiceImpl	src
BO	acube.projet.business.bo.<cas d'utilisation>.nomBO	acube.projet.business.bo.DossierBO	src
DAO (interface)	acube.projet.integration.dao.nomTableDAO		src
DAO (implémentation)	acube.projet.integration.dao.nomTableDAOImpl		src
DAO (mapping SQL)	acube.projet.integration.dao.map.nomTable_sqlMap.xml		src
Classe de test d'action	acube.projet.web.action.<cas d'utilisation>.nomActionTest		tst
Classe de test de service	acube.projet.business.service.<cas d'utilisation>.nomServiceTest		tst
Template (JSP ou XSL)	templates/<cas d'utilisation>/nomAction.jsp (.xsl)		WebContent
Log4J	log4j.properties		src
Struts Config	struts.xml		src
Spring général + business	appContext.xml		src
Spring dao	appContext-dao.xml		src
Spring Security	appcontext-security.xml		src
IBatis SqlMapConfig	ibatis.xml		src

5 OUTILS DE DEVELOPPEMENT

- Eclipse (avec template de dev java)
- Plugins : Checkstyle, Findbugs, PMD, iBator (abator)
- Firefox + FireBug pour le code html + javascript

FIN DU DOCUMENT