Java/J2EE

# Présentation

Bienvenue sur le Wiki J2EE Open Source de Mediapost !

Vous y trouverez une présentation des outils, normes, méthodes et concepts à mettre en oeuvre pour toute nouvelle réalisation d'une application J2EE Open Source au sein de la DSI de Mediapost.

## 

Vous trouverez ci-dessous la liste des URL des différents outils mis à votre disposition :

* **Le Wiki** (celui que vous êtes en train de lire !) se situe sur [http://http://appsiege.mediapost.fr:98/confluence](http://http//appsiege.mediapost.fr:98/confluence)
* **Le Repository Subversion** : le repository Subversion sur lequel doit pointer Subclipse est à l'adresse <http://subversion.mediapost.fr/svn/svnroot>
* **XPlanner** : l'application de gestion des users stories se situe sur <http://testunit.mediapost.fr:8080/xplanner>
* **L'Intégration Continue** : l'application Continuum qui gère le process d'intégration continue se trouve sur <http://sunic.mediapost.fr:8080/continuum>
* **Archiva** : le repository mirroir pour Maven est situé à l'adresse <http://sunic.mediapost.fr:9998/archiva>
* **Sonar** : le reporting Sonar est situé à l'adresse [http://sunic.mediapost.fr:9000](http://sunic.mediapost.fr:9000/)

# Architecture Générale applications Web

Ce chapitre constitue un guide d’architecture pour le développement d’applications Web en Java, conforme à la spécification JEE et aux bonnes pratiques recommandées par Sun à travers ses guides d’architecture *BluePrints* et ses *Design Patterns Reviews* mais aussi par d’importants acteurs tels que *Martin Fowler* et le Gang of Four (*Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides*).

Les briques logicielles référencées dans ce document s’appuient sur le socle commun constitué par la spécification JEE.

Ce document a pour objectifs de définir les relations inter composants et de spécifier les traitements types sous-jacents au schéma de l’architecture applicative définie pour Mediapost. Ce dossier présente également les principes d’implémentation des différentes couches de l’architecture applicative décrites dans les [préconisations en terme de design](#scroll-bookmark-4).

La mise en place du socle technique Mediapost a pris en compte les contraintes qualité suivantes :

* La facilité de maintenance des règles fonctionnelles souvent complexes,
* La maîtrise de l’architecture et de la technologie choisie,
* La résistance aux fortes et courtes montées en charge,
* La scalabilité et la robustesse des applications,
* La pérennité des choix.

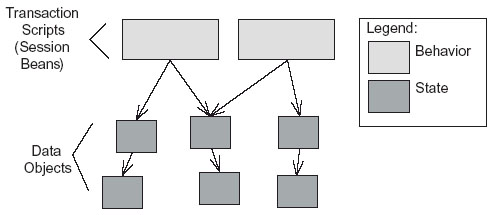
## Glossaire Java/J2EE

|  |  |  |
| --- | --- | --- |
| **Terme** | **Abréviation** | **Définition** |
| Couche |  | Ensemble d’éléments concourants à l’application de rôles et services communs. |
| Spécialisation |  | La spécialisation d’une couche est la séparation d’une ou plusieurs problématiques/rôles particuliers des autres rôles de la couche. La spécialisation au sens large est la déclinaison d’un thème, d’une particularité à partir d’un thème général : L’oiseau est une spécialisation de la notion d’animal. |
| Service |  | Composant accessible depuis une couche ou une autre application. |
| Logique applicative |  | Algorithme général d’un traitement contenant un enchaînement de règles de gestion et autres traitements. |
| Règle de gestion |  | Traitement élémentaire d’un algorithme : un contrôle (contrôle du poids), un calcul, une préparation de données, une recherche d’information. |
| Système d’exploitation | OS | Operating System. Système d’exploitation. |
| J2SE | Plateforme Java 2 Standard Edition. | Version standard de la plate-forme Java version 2 qui dispose de nombreuses fonctionnalités de base : JDK, JFC-Swing, applets, Awt, JDBC, RMI, JNDI, …. |
| J2EE ou JEE | Plateforme Java 2 Enterprise Edition | Version « Enterprise » de la plate-forme Java 2 qui se compose de l’environnement J2SE plus de nombreuses API et composants destinés à une utilisation côté serveur (servlet, JSP, EJB, JCA, JMS, JTA, …). Dénommée JEE depuis sa version 5 |
| J2ME | Java 2 Micro Edition | Version hautement optimisée de la plate-forme Java version 2 destinée à des applications embarquées : téléphones portables, « pagers », systèmes de navigation, … |
| JRE |  | Java Runtime Environment |
| Base de données relationnelles | SGBDR | Système de Gestion de Base de Données Relationnelles |
| HTTP | Hyper Text Transfer Protocol. | Protocole standard de transfert de fichiers hypertexte basé sur TCP/IP. |
| HTML | Hyper Text Markup Language. | Langage de présentation dérivé du SGML basé sur un jeu de balises (ou « tags ») qui définissent la présentation du document. |
| Javascript | JS | Langage de script interprété utilisé dans les pages HTML. « ECMAscript » est la version standardisée. |
| Java Database Connectivity | JDBC | Middleware permettant l’accès d’une application écrite en JAVA à une base de données relationnelles. |
| Java Data Objects | JDO | Il s’agit d’une spécification qui vise à fournir une gestion de persistance totalement transparente au développeur et objet. |
| BluePrint |  | Les « Blueprints » (littéralement « croquis » ou, « plan ») sont destinés à aider les développeurs Java en leur fournissant tous types d’informations utiles : des conseils, des modèles de conception, des exemples de codes, des articles de presse, des Q/R, etc… |
| Conteneur (container) |  | Il s’agit de l’environnement d’exécution contenant les différents types de composants de la plate-forme Java. |
| Introspection/réflection |  | Capacité de Java permettant de découvrir dynamiquement les attributs et les méthodes d’une classe pendant l’exécution. |
| Sérialisation |  | Mécanisme standard offert par Java permettant d’enregistrer le contenu d’un objet dans un flux (souvent un fichier) puis de le reconstituer ultérieurement. |
| Package |  | Un package est un ensemble de classes. Ils permettent d’organiser les classes en composants logiciels cohérents. |
| Persistance |  | Capacité des objets à perdurer indépendamment de leur environnement d’exécution. Il s’agit de la possibilité pour un objet de se sauvegarder sur disque afin d’être reconstitué à l’identique ultérieurement. |
| Proxy |  | Objet technique dont le rôle est de simuler localement un objet distant. |
| Serveur d’application |  | Serveur logique fournissant un ensemble de services applicatifs et qui assure les interactions entre les postes des utilisateurs et les différentes ressources du système d’information. En Java, ces serveurs sont normalisés et doivent être conforme aux spécifications J2EE. |
| Service Web |  | Service distant invocable par le Web en utilisant le protocole SOAP, les annuaires UDDI et le langage de description WSDL. |

## Préconisations en terme de design

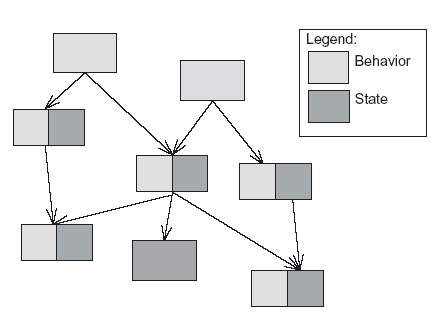
Une approche **orientée objet** (à l’opposé d’une approche procédurale) est préconisée pour les applications JEE Mediapost. Cette approche orientée objet est à privilégier et est particulièrement adaptée lorsque les architectures sont à base de POJO (*Plain Ordinary Java Object* – objet java simple) et utilisent des conteneurs légers, tels que Spring.

L’approche procédurale typique est incarnée par une architecture à base d’EJB 2. Martin Fowler a décrit cette approche procédurale à travers le pattern *Transaction Script*. Selon ce pattern, plutôt que d’utiliser un design orienté objet, il suffit d’écrire une méthode, implémentant un *transaction script*, qui traite chaque requête de la couche présentation. Selon cette approche, les classes qui implémentent le comportement sont séparées de celles qui stockent l’état. Une des difficultés de cette approche provient du fait que le comportement est concentré dans peu de classes « larges », le code devient alors difficile à comprendre et à maintenir. Martin Fowler a décrit les risques de ce type d’architecture à travers l’anti-pattern «Anemic Domain Model».



**La structure d'un design procédural : de larges *transaction script* classes et beaucoup de petits *data objects***

Dans l’approche orientée objet, la logique métier est représentée par un modèle objet, qui est un agencement de petites classes correspondant à des concepts du domaine métier. Une application bancaire par exemple pourrait, en suivant cette approche, être implémentée à base d’objets tels que TransferService, Account, BankingTransaction. Ces classes peuvent embarquées du comportement et un état. Martin Fowler a décrit cette approche orientée objet à travers le pattern *Domain Model*. Ces objets peuvent être appelés «objets domaine», «objets métiers» ou «entités métiers», nous privilégierons la terminologie **objets domaine** pour ce type d’entités.



**La structure d'un *domain model* : des petites classes qui ont des données et du comportement**

Il existe un grand nombre d’applications ne respectant pas ce principe. Classiquement au sein de ces dernières, on rencontre des objets services qui se voient injectées des objets DAO et qui utilisent ces DAO pour gérer la persistance des objets domaines. Les objets domaine eux-mêmes ne sont alors que de simples transporteurs de données. Le problème avec cette approche est que l’interaction entre les objets de la couche service et les objets DAO devient très procédurale, à l’antithèse d’une véritable approche objet. L’objet service effectue généralement un ou plusieurs appels vers les DAO en leur passant l’objet domaine qui ne fait que stocker les données. On peut ainsi tomber dans les pièges du pattern *transaction script* de l’approche procédurale décrit par Martin Fowler.

Un meilleur design est de laisser **l’objet domaine gérer du comportement, notamment des calculs ou contrôles sur son état** (ses propres attributs, ses objets agrégats) jusqu’à des traitements tels que ceux nécessaires à sa propre persistance. Selon la préconisation consistant à envisager un objet domaine déléguant à un DAO la gestion de sa propre persistance, un problème réside dans le fait que ce type d’objet n’est classiquement pas géré par un containeur léger. Comment Spring peut-il injecter un DAO dans un objet domaine lorsqu’il n’instancie pas lui-même cet objet domaine ? En effet, ces objets domaines sont classiquement instanciés en dehors de Spring, en l’occurrence par Hibernate ou un autre mécanisme de persistance. Plusieurs solutions sont envisageables. La solution préconisée par Spring utilise les annotations de Java 5, elle est basée sur AspectJ et est disponible avec la version 2.0 de Spring (elle n’a pas été testée chez Mediapost). Même si l’objet domaine ne va pas jusqu’à encapsuler des traitements gérant sa persistance, l’idée est de positionner **un maximum de méthodes de comportements au sein de cet objet domaine**, afin de s’éloigner ainsi du modèle anemic selon lequel ces types d’objet ne sont que des convoyeurs de données.

Selon ce design, les objets service peuvent ainsi directement interagir avec les objets domaines en respectant une approche beaucoup plus « objet ». Au lieu de sous-traiter au DAO un traitement de validation particulier sur d’objet Client par exemple, le service peut utiliser directement des méthodes dédiées de l’objet Client lui-même.

## Architecture multi-couche

L’objectif est de découper une application en plusieurs couches logiques spécialisées. Cette section définit :

* Une terminologie commune pour ces couches logicielles.
* Les responsabilités et services inhérents à chaque couche.
* Des choix en termes de framework ou de design pattern adaptés à chaque couche.

### Objets du domaine et bloc fonctionnel

Les objets du domaine sont les données métier manipulées par l’application, elles sont soit persistantes par le biais de la base de données, soit « transient ». Les objets du domaine peuvent implémenter des traitements tant que ceux-ci ne concernent que l’objet et ses agrégats. Les objets du domaine n’accèdent pas aux services techniques tels que la base de données. Il s’agit de POJO (Plain Ordinary Java Object), mappés pour ceux qui sont persistants avec une table de la base de données. Ils traversent sans limitation les couches. Les objets du domaine sont donc utilisés pour communiquer entre toutes les couches. Ce ne sont pas des singletons, ils sont *stateful* (avec état) et non *threadsafe* (accès par plusieurs threads non géré). Ils sont utilisés comme paramètre entrée/sortie des méthodes de services (business Deleguate, métier ou DAO). Ils peuvent également être appelés directement dans ces mêmes services, étant donné qu’ils encapsulent certains traitements (en adéquation avec une approche « orientée objet » à l’opposé de « l’Anemic Model ». En aucun cas ils ne doivent constituer l’état d’objets services métier ou DAO qui, eux, sont des singletons *stateless* (sans état).

Les données et certains traitements de l’application sont ainsi structurés en **objets domaine** eux-mêmes regroupés en domaines métiers (exemples de domaines métiers : Clients, Matériel, Société, etc.)

Un domaine métier peut être utilisé par différentes applications et n’est pas lié à un domaine applicatif particulier. Il peut correspondre à un bloc fonctionnel, mais il est courant qu’une fonctionnalité de l’application s’appuie sur plusieurs domaines métiers.

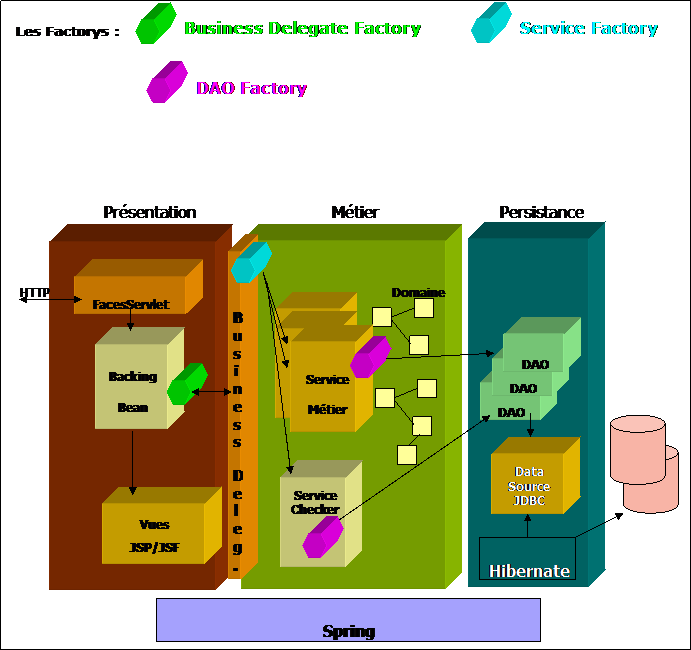
Les différents domaines métiers sont, dans la mesure du possible, conçus pour que les interactions entre eux soient minimales. Les associations sont courantes entre les objets de différents domaines métier, mais les objets ont des visions restreintes des objets associés. Cette restriction doit être renforcée par l’utilisation d’interfaces.

Les fonctionnalités, implémentées à partir des cas d’utilisation, forment des blocs fonctionnels. Un bloc fonctionnel regroupe des fonctionnalités utilisées souvent conjointement et s’appuyant sur un ensemble de domaines métier (Exemple : saisie et validation des demandes de congés s’appuie sur les domaines métier Société, Collaborateur, Planning, etc.)

Les cas d’utilisation peuvent être regroupés en blocs fonctionnels.

### Présentation du modèle en couches

Le schéma suivant liste les différentes couches logicielles d’une application, depuis l’interface graphique jusqu’aux entrepôts de stockage de données :



**Les différentes couches logicielles**

Voici de manière résumée les responsabilités de chaque couche :

* **La couche Présentation**est implémentée avec JSF (implémentation MYFACES). Elle est découpée en deux parties :
  + **L'Interface graphique**contient les composants graphiques permettant la construction des écrans de l’application :
    - Menus, formulaires, zones de saisies…
    - Editions
    - Listes multi-pages…
    - Validation premier niveau des entrées utilisateurs.
    - Gestion du type de média : Web, Palm, etc…
  + **La Logique Applicative**est implémentée en partie dans le front Controleur (Faces Servlet) et en partie dans les backing beans JSF gère :
    - La cinématique et la navigation dans l’application.
    - Contrôles de validation de premier niveau spécifiques à la page.
    - Gestion du cycle de vie des objets en session.
    - Formatage des champs saisis ou affichés.
    - La localisation et la délégation aux composants de la couche Métier pour les processus de traitement et de validation « métier »
    - La gestion de la session utilisateur et du contexte.

Les Backing Beans sont classiquement des objets stateful, ce ne sont pas des singletons. Ils correspondent classiquement à une instance de formulaire HTML.

* **La couche Métier** a pour fonction d’implémenter les règles métier, d’assurer le dialogue avec la couche de persistance. La couche masque la complexité métier (règles de gestion) et technique (accès aux données par exemple) à la couche présentation qui ne manipule le métier que par le biais d’objets Java (POJOs) et des méthodes des Business Delegate. La couche métier est constituée des composants suivants :
  + **Business Delegate (façades)** : les façades sont le point d’entrée obligé pour la couche présentation pour accéder au métier, bien que faisant partie du métier, les façades sont conçues pour être utilisé dans le cadre d’une fonctionnalité donnée (ou d’un groupe de fonctionnalités). Une méthode d’un Business Delegate appelle une ou plusieurs méthodes d’un nombre variable de services ou de méthodes d’objets domaine. Les Business Delegate peuvent être transactionnels.
  + **Service Métier** : des traitements, par exemple certaines vérifications, sont nécessaires sans pouvoir être implémentés dans les objets du domaine (impliquant par exemple plusieurs objets du domaine). Les services métiers permettent d’implémenter ce type de traitements. La gestion transactionnelle est également gérée au niveau des services métier (via des proxy Spring de ces services métier). Ainsi lorsqu’un service appelle une méthode d’un DAO ou une méthode de traitement d’un objet domaine, il est possible d’encapsuler ces traitements au sein d’une seule et même transaction.

Cette couche a la responsabilité :

* De la gestion des validations et processus fonctionnels (via les Business Delegate)
* De l’implémentation des règles de gestion métier (de contrôle, de calcul, de préparation des données à mettre à jour).
* De l’intégrité transactionnelle (début, validation, annulation) des données applicatives (mises à jour des données préparées) : Orchestration d’appels à des DAO pour la récupération/mise à jour de données

Les services sont créés via Spring, ils ont les caractéristiques suivantes :

* Il s’agit de singleton
* La gestion des transactions est faite à ce niveau. Elle est déclarative.

Les services sont des objets stateless, threadsafe.

* **La couche Persistance**représente les objets des sources de données et éventuellement de leurs relations. La couche de persistance permet la sauvegarde des données sur un support physique ainsi que leur restitution. Les applications pourront s’appuyer sur le moteur de requêtage de la base de données, parfois plus approprié que la récupération de l’intégralité de données dans le serveur d’applications. Par ailleurs, les applications s’appuient sur le moteur de base de données pour la gestion des transactions. A l’exception des contraintes d’intégrité référentielles, la couche de persistance n’implémente pas de règles de gestion (sous forme de trigger, de procédure stockées ou n’importe quel autre moyen proposé par le moteur de base de données). La persistance s’appuie en principe sur une base de données relationnelle (ce qu’il n’exclut pas d’autres modes de persistance : fichier XML, B-Tree, base de données orientée objet). Cette couche à la responsabilité :
  + Des traitements de mise à jour des entités métier (create, read, update, delete).
  + Des traitements de récupération de données (requêtes de catalogues, de recherches…).
  + Du maintien de la cohérence transactionnelle des processus lancés par la couche Métier. La transaction est démarrée, commitée ou rollbackée au sein du service (service métier ou Business Delegate) et non pas au niveau du DAO qui, lui, ne fait que s’exécuter au sein d’un contexte transactionnel que le service a initié.
  + Du mapping Objet / Relationnel.
  + Des Listeners : Le mécanisme d’événement de Hibernate peut être utilisé pour implémenter des règles métier telles que certaines validations. Les validations implémentées de cette manière présentent l’avantage de ne pas nécessiter d’appel explicite à une méthode : la vérification est effectuée automatiquement au flush de la session Hibernate.

Il existe également les couches middleware et physiques qui sont sous la responsabilité la plupart du temps des éditeurs : drivers JDBC et base de données, pilotes JNDI et annuaire LDAP…

### Principes généraux d'architecture

L’intérêt même d’un modèle d’architecture en couches réside dans l’indépendance des couches entre elles. Cela permet de disposer d’un système évolutif (changement de base de données ou d’outil de mapping objet / relationnel sans impacter les autres couches par exemple), et de pouvoir développer et tester séparément les couches.

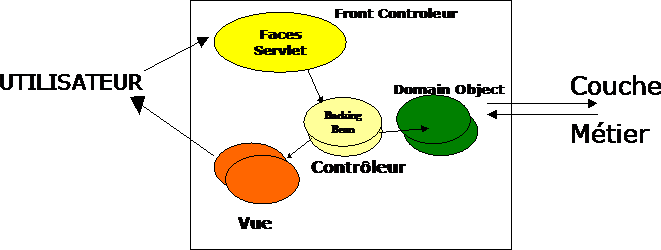
Cela repose toutefois sur l’utilisation de Design Pattern éprouvés dans la structuration des couches et la communication inter-couches :

* MVC2 pour la Couche Présentation et modèle événementiel apporté par JSF
* Inversion de contrôle (Injection de dépendances) de façon transverse.
* Les factory
* Data Access Objects pour la Couche Persistance.
* Value Object (encore appelé Data Transfert Object ou view)

#### Modèle / Vue / Contrôleur (Couche Présentation)

* **La Vue** est (dans le cadre d’application web) le plus souvent implémentée par des pages JSP, afin de construire dynamiquement du HTML. Ces JSP vont inclure des taglibs (bibliothèques de tags) issues du framework JSF.
* **Le Modèle** est un objet qui va permettre de stocker le domaine métier de l’entreprise
* **Le Contrôleur** fait le lien entre la Vue et le Modèle, en gérant l’enchaînement de la Navigation et le traitement des actions/événements d’un écran. Selon la spécification JSF, le contrôleur est implémenté dans les méthodes actions des backing beans. Un Front Contrôleur est implémentée par la servlet de JSF pour la navigation et le cycle de vie d’une requête JSF.

Cela est résumé dans le schéma suivant :



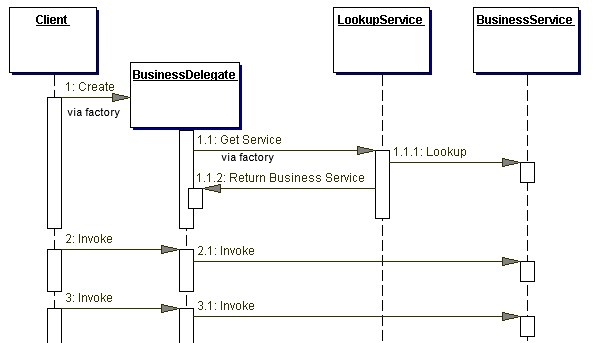
**Modèle MVC**

La communication entre le backing bean et le service métier se fait par l’intermédiaire d’un délégué (pattern Business Delegate).

Cela permet de :

* réduire le couplage entre la couche Présentation et la couche Métier.
* masquer à la couche Présentation la logique de localisation et d’instanciation des services de la couche Métier.

Le schéma suivant illustre le fonctionnement d’un délégué :



#### Inversion de contrôle

L’inversion de contrôle répond à un besoin simple : limiter (dans l’idéal supprimer) la dépendance d’une couche vis-à-vis de l’implémentation de sa sous-couche. L’intérêt est de pouvoir développer et tester séparément les couches, voire changer l’implémentation d’une couche sans que ce soit trop coûteux.

La solution revient à utiliser dans la couche appelante une interface de la couche N-1 (la couche appelée). L’interface doit être indépendante de la technologie employée pour implémenter le service et être la plus fonctionnelle possible.

Comment réaliser alors l’instanciation proprement dite de l’implémentation de l’interface dans la couche appelante ? Par l’utilisation d’un framework d’injection de dépendances (conteneur léger) : on injecte ainsi au service de la couche appelante la dépendance (un autre objet de la couche appelée) dont elle a besoin.

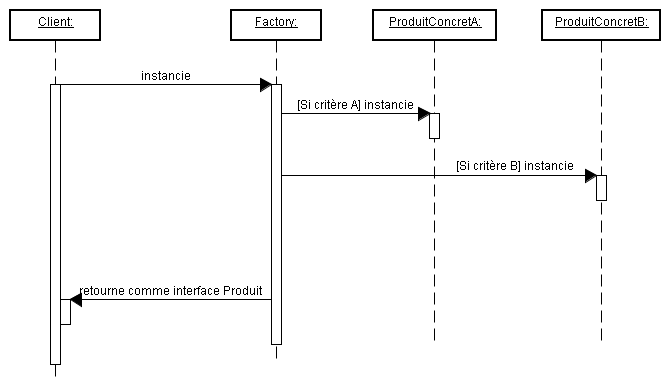
Le conteneur a la responsabilité de charger les composants et de gérer les relations (dépendance) entre ces composants. Cela implique que les composants (et leurs relations) doivent être déclarés au niveau d’un fichier de configuration du conteneur.

Spring est un framework de type conteneur léger par exemple. Il offre en plus des fonctionnalités intéressantes telles que la gestion déclarative des transactions. Il est implémenté au sein de l’architecture Mediapost.

#### Les factory

L’invocation d’un service de la couche coordination se fait via une **Business Delegate Factory**, cette invocation est faite via la classe mère de toutes les Backing Beans JSF. La couche présentation ne manipule alors que des interfaces de **Business Delegate**.

Le schéma suivant illustre le fonctionnement d’une factory :



On retrouve dans l’architecture cible de Mediapost des factorys pour la communication inter couches des objets traitements :

* Les backing beans appellent un Business Delegate via une Business Delegate Factory
* Les Business Delegate appellent un service metier via une Service Factory
* Les Service metier appellent un DAO via une DAO Factory

Au sein de l’architecture Mediapost, il est préconisé que chaque factory soit déclarée dans Spring en tant que bean singletons : on spécifie leurs dépendances au sein du fichier de configuration de Spring. Une DAO Factory, par exemple, se verra injecter des implémentations spécifiques de DAO. Elle disposera alors de méthodes explicites pour chacun de ses DAO du type getClientDAO(), getFournisseurDAO(), etc.

#### Data Access Objects

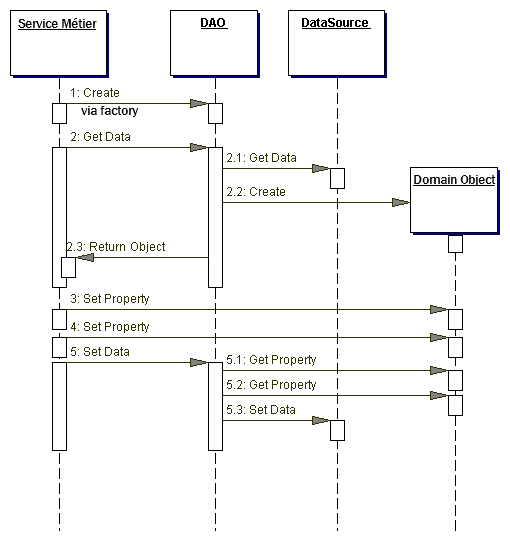
L’idée est d’encapsuler dans une couche dédiée tous les accès à l’entrepôt persistant de données. Ce pattern concerne donc la **Couche Persistance**.

Le principe général d’indépendance entre couches s’applique évidemment entre la couche Métier et la couche Persistance, donc la localisation et / ou l’instanciation d’une DAO doit se faire via son interface.

Le diagramme suivant, issu du site de référence SUN (<http://java.sun.com/blueprints>) illustre ce mode de fonctionnement.

Ajoutons quelques compléments pour la compréhension de ce diagramme de séquence :

* le service Métier se voit injecter une DAOFactory. Il peut alors, via cette DAOFactory, récupérer une implémentation d’un DAO via une méthode explicite du type (daoFactory.getCLientDAO()).
* le service métier peut alors appeler une méthode spécifique sur ce DAO (GetData() dans l’exemple ci-dessous). Une Datasource est injectée au DAO (cette injection est spécifiée au sein du fichier de configuration de Spring). Le fait que le DAO sous-traite la récupération des données à sa datasource et créé un objet domaine est masqué au développeur grâce à l’utilisation d’Hibernate.
* le DAO retourne pour finir l’objet domaine au service métier qui va pouvoir le manipuler directement.



Les implémentations des DAO peuvent fonctionner de différentes manières :

* Accès direct à la base et lancement d’ordres SQL via JDBC (avec le framework DAO Spring JDBC)
* Appels à une couche de mapping objet / relationnel issue du framework Hibernate.

Dans les deux cas, les services de la Couche Persistance doit pouvoir s’insérer dans les contextes transactionnels initiés (de manière déclarative ou programmative) par la Couche Métier.

La couche DAO encapsule les opérations sur les données (récupération, mise à jour de la base de données). Pour Mediapost, le moteur de mapping objet/relationnel choisi est Hibernate. L’utilisation d’Hibernate est complètement cantonnée à la couche persistance, aucune autre couche ne doit y faire référence.

Par défaut, les DAO héritent d’une classe du framework Hibernate DAO de Spring (HibernateDAOSupport) et utilise donc Hibernate.

Au sein de l’architecture Mediapost, chaque DAO héritent d’une classe fr.mediapost.fwk.integration.hibernate.AbstractHibernateDAO qui elle-même hérite de HibernateDAOSupport (voir un exemple d’implémentation d’un DAO dans le prototype MuncipostPos : fr.mediapost.municipost.integration.impl.HibernateClientDAO).

Pour des traitements TRES spécifiques, s’il est nécessaire de passer directement en SQL, il faut utiliser de la même manière le support de DAO SQL de Spring (classe mère et JdbcTemplate). Cette homogénéité permet notamment de conserver la même politique de gestion des transactions avec des DAO hybrides.

Pour la récupération d’objets, Hibernate propose plusieurs moyens. Chaque développeur a ses préférences, mais dans un souci d’uniformité, il faut se tenir au possible à la politique suivante :

* Utilisation de l’API Criteria par défaut, pour les requêtes basiques (opérateurs AND dominants)
* Utilisation de HQL pour les requêtes plus complexes (opérateurs AND/OR mêlés, jointures complexes, requêtes dites de rapports par exemple SUM, AVG)

Cette politique est choisie notamment pour les raisons suivantes :

* API Criteria suffisante pour la plupart des requêtes
* Moins de manipulations de chaînes de caractères

Si l’application comporte des calculs, ne pas hésiter à utiliser la base de données pour effectuer certains d’entre eux, plutôt que de le faire en Java dans la couche métier. Cela peut fortement influencer les performances.

Les DAO ne peuvent être utilisés qu’au sein de la couche métier (et entre DAO) et surtout ne doivent jamais être appelés directement par la couche présentation.

L’objectif est que les DAO n’ouvrent pas une connexion à la base de données à chaque ordre SQL. Les connections doivent être poolées dans un souci de performance.

Dans le cas de Mediapost, la configuration des connexions se fait via Spring. Il est donc aisé de passer d’une implémentation à une autre, selon le besoin :

* Application mono datasource (une seule base de données) : utilisation de C3PO
* Application multi datasources (plusieurs bases de données) : utilisation de Enhydra XA Pool
* Tests unitaires : pas de pooling nécessaire, utilisation de la classe SingleConnexionDataSource de Spring

#### Data Transfert Objects

Dans certains cas, un objet du domaine ne suffit pas pour transmettre toutes les informations nécessaires, par exemple pour représenter tous les critères sur lesquelles peut porter une recherche. Il peut alors être utile de définir un DTO (*Data Transfert Object*) qui peut, par exemple, être un agrégat d’entités. Cela évite notamment de définir des méthodes avec de nombreux paramètres.

Le pattern DTO doit cependant être utilisé avec parcimonie puisque chaque DTO est fortement lié aux entités qu’il agrège et que toute modification sur l’un deux doit aussi être répercutée.

Le principe est de centraliser les données métiers homogènes entre elles concernant un domaine fonctionnel ou un besoin ou un cas d’utilisation dans une structure de données simple.

Un DTO (ou value object) représente soit un agrégat d’objets du domaine (produit, commande ou commande et lignes de commande), soit un bloc visuel de la page Web (un onglet, un ensemble de champs, les données de la page entière). L’implémentation d’un DTO est une simple classe possédant des getter/setter

Des conteneurs de DTO peuvent s’avérer utiles pour gérer une liste d’enregistrements, par exemple :

Classe liste de Commandes :

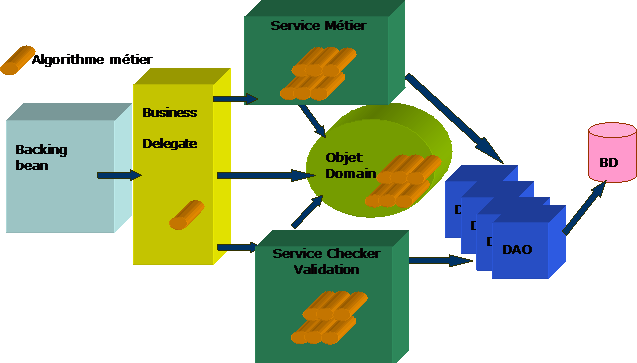
public class OrdersList implements Serializable {  
  
 public int size() {…};  
 public boolean contains(Object o) {…};  
 public boolean containsAll(Collection c) {…};  
 public boolean equals(Object o) {…};  
 public boolean isEmpty(){…};  
  
 private OrdreDetail[] listOfOrderDetail;  
  
}

### Dissémination du métier

L’objectif est de concentrer le métier dans les objets dédiés : les objets du domaine pour les traitements de validation typiquement, les services métier permettant de stocker les règles métier. Il peut être intéressant d’effectuer les validations portant sur plusieurs entités métiers dans les services métier dédiés : des checkers.

**Il peut être également possible de s’appuyer plus grandement sur les listeners Hibernate pour effectuer des contrôles techniques. Il peut être également intéressant de s’appuyer sur Hibernate Validator pour certains contrôles dédiés à la validation des objets domaine.**

Le métier contenu dans le Business Delegate concerne l’aspect BPM (Business Process Management), c’est-à-dire l’agrégation et l’orchestration des processus métier.



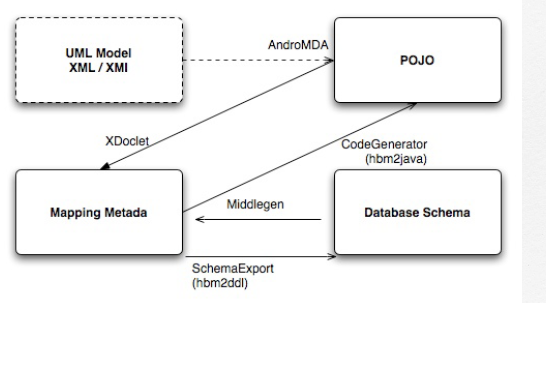
**Dissémination du métier**

### La génération de code et le danger de l'approche DDA

Plusieurs outils peuvent être utilisés pour générer les POJO Hibernate ou les fichiers de mapping (voir schéma ci-dessous) :

* Dans une approche MDA (basée sur les modèles objets, UML en l’occurrence) : AndroMDA permet de générer les POJO mais également grâce à son concept de cartbridge les DAO, services métiers, backing beans, etc.) et Xdoclet pour les fichiers de mapping
* Dans une approche DDA (basée sur les données, le schéma de la base de données l’occurrence) : middlegen pour les fichiers de mapping et hbm2java pour les POJO

Nous préconisons une approche MDA, plus proche de la philosophie de l’objet, néanmoins il est parfois difficile sur un projet de ne pas être contraint par un modèle de données existant.



**Approche MDA**

Lorsque la base de données existe déjà la génération de code est souvent attirante pour démarrer rapidement le projet. Malheureusement, les bases de données s'appuient généralement sur des conventions de nommage qui n'ont rien à voir avec le code objet et java notamment. Il est très rarement judicieux de garder les mêmes noms de colonnes pour les propriétés java.

Exemple :

Un modèle de données contient des tables de références comme la table des pays, la table des sites, ...

Les colonnes de ces 2 tables sont :PAYS : CODE\_PAYS, LIB\_PAYSSITE : CODE\_SITE, LIB\_SITE

Dans le meilleur des cas, une génération automatique du mapping et des POJOs va donner le résultat suivant :Classe Pays : propriétés codePays, libPaysClasse Site : propriétés codeSite, libSite.

Le problème de cette génération est qu'elle empêche toute la réutilisation de code que permettent les mécanismes objets. Les deux classes sont en effet trop différentes pour s'appuyer sur des mécanismes communs.

Alors que si ces 2 classes avaient été mappées de la façon suivante :Classe Pays : propriétés code, libelleClasse Site : propriétés code, libelle.

Elles auraient pu implémenter une interface commune.

Il devient alors possible d’utiliser les mêmes jsp pour lister, consulter, éditer ces classes. En utilisant l'api d'Hibernate, il est possible de fournir un traitement commun de ces classes pour leur recherche, chargement et modification. Cela engendre ainsi parfois beaucoup moins de duplication.

En conclusion, lorsque le mapping a été généré depuis une base existante, il est très important de prendre le temps de bien comprendre le modèle de donnée et de rechercher les similitudes avant de paramétrer la génération.

### Framework technique Mediapost

Un projet, regroupant différents composants techniques réutilisables, a été créé. Les applications et bibliothèques métier pourront s’appuyer sur ce socle et l’étoffer au fur et à mesure des développements. Ce framework reste technique et n’a pas pour objectif d’imposer l’architecture d’une application. Il contient notamment des classes de base pour les tests unitaires, des utilitaires de validation, des classes communes (classe d’exception métier)…

## Services transversaux

Certains services transversaux aux différentes couches applicatives et communs à toutes applications doivent êtres gérés.

### L'authentification et les autorisations

Un service doit être mis en place afin de contrôler que l’accès à chaque page de l’application est contraint à une préalable authentification (LDAP, DB) de l’utilisateur.

Cela peut être réalisé de manière déclarative (Realm Tomcat par exemple), ou programmative. Il faut dans ce cas veiller à ce que le code ne soit pas dupliqué : héritage de contrôleurs, filtre http…

Voir [Comment gérer la sécurité](#scroll-bookmark-20) pour la solution minimaliste implémentée dans le framework via un ActionListener JSF. Nous préconisons d’étudier une solution plus complète comme ACEGI (voir [Une rapide présentation d'ACEGI](#scroll-bookmark-21) pour une présentation de cette solution).

### La gestion des exceptions

Le but de ce service est de fournir un mécanisme standard de remontée des exceptions techniques et métier à travers les différentes couches de l’application.

En dernier lieu, ce service doit assurer la détermination du message et la redirection de l’utilisateur vers une page d’erreur standard, et la trace de l’incident survenu.

Voir [Gestion des exceptions](#scroll-bookmark-23) pour la description de la solution proposée dans le Framework Mediapost.

### La journalisation

Le but de ce service est de fournir une API uniforme pour les processus de journalisation, à travers les différentes couches de l’application.

Toutes les applications ont des besoins de journaliser des informations sur leurs états suivants des niveaux différents (Debug, Info, Warning, Error). Il est souhaitable de pouvoir contrôler ce niveau de journalisation de manière déclarative afin de pouvoir passer du niveau Debug (utile lors de la phase de mise au point et recette) au niveau souhaitable en production (Warning par exemple) qui ne laissera passer que les messages associés à ce niveau ou supérieur (Warning et Error par exemple). Cela permet d’éviter de journaliser des messages de moindre importance et donc, d’améliorer les performances.

LOG4J est un framework éprouvé concernant la journalisation. Il distingue 4 niveaux de trace :

* Le type DEBUG à pour objectifs la mise au point du développement et doit donc pouvoir être débrayé complètement en production
* Le niveau INFO a pour objectif de fournir des informations de suivi en production à la demande ou systématiquement. Il est donc en général désactivé mais peut, pendant les phases de déploiement ou pour détecter un problème particulier, être activé en production.
* Les niveaux WARNING et ERROR sont utilisés en production pour alerter ou conserver une trace des dysfonctionnements rencontrés.

### Les tests unitaires

L’architecture applicative doit permettre de pouvoir tester unitairement et indépendamment les Couches Métier et Persistance.

* Couche métier : utilisation de mock object pour la couche de DAO
* Couche persistance : utilisation de DbUnit pour mettre dans un état connu la base de données avant chaque test unitaire, cela grâce à des fichiers de données XML (dataset)

Le framework Mediapost contient des classes utilitaires pour la mise en place des tests unitaires. Le chapitre [Framework Mediapost](#scroll-bookmark-26) et le prototype Municipost approfondissent ces explications avec des exemples.

### La gestion des transactions

Les transactions sont gérées de manière déclarative avec Spring.

Voir [Gestion des transactions](#scroll-bookmark-28) pour la description détaillée de la solution proposée au sein de l’architecture Mediapost.

### La concurrence d'accès

Il s’agit d’éviter que sur deux utilisateurs éditant simultanément la même donnée, les valeurs du deuxième « écrasent » les valeurs du premier. Différentes stratégies existent classiquement pour gérer ce problème d’isolation :

* 1ère solution : ne pas la gérer, si le risque est considéré comme acceptable, en tout cas non critique.
* 2ème solution : Gestion optimiste -> gérer un champ spécial dans les tables (un champ timestamp ou un numéro de version), mis à jour lors de chaque update, et contrôlé AVANT de réaliser l’update. Hibernate peut gérer ce mécanisme : il incrémente ce numéro à chaque mise à jour et il se charge de comparer le champ version d’un enregistrement à mettre à jour avec le champ version de l’enregistrement dans la base. Si l’enregistrement en base est plus récent, Hibernate lance une exception. Au programmeur de la gérer.
* 3ème solution : Gestion pessimiste -> utiliser un système de lock des données. Au moment de la lecture de l’information, les données lues sont lockées jusqu’à fin de la transaction.

La politique de gestion de la concurrence choisie pour Mediapost est optimiste puisqu’elle n’est pas basée sur un lock des données. Elle est similaire à la deuxième solution, mais ce n’est pas Hibernate qui va gérer la mise à jour du champ spécial. La solution Mediapost est basée sur un champ timestamp au sens Sybase, mis à jour directement par le moteur Sybase. Hibernate va se contenter de lire ce champ.

Voir [Concurrence d'accès](#scroll-bookmark-30) pour la description de la solution proposée dans le Framework Mediapost.

## Composants Java/J2EE

Le tableau suivant résume les choix d’architecture faits pour les points cités précédemment :

|  |  |  |
| --- | --- | --- |
| **Point** | **Solutions/versions** | **Remarque** |
| Système d’exploitation | Windows |  |
| Conteneur Web | Tomcat 5.5 |  |
| Machine Virtuelle Java | Sun 1.5 |  |
| Base de données | Sybase |  |
| Couche Interface graphique | JSP 2.0 ou Facelets 1.1.11, Taglib MyFaces & TomaHawk |  |
| Couche Logique Applicative | Myfaces 1.1.4 |  |
| Couche Données Métier | Hibernate 3.2 |  |
| Inversion de contrôle | Spring 2.0 |  |
| Journalisation | Log4j 1.2.9, Jakarta Commons Logging 1.0.4 |  |
| Pooling de connexions | C3PO 0.9.04, XA Pool 1.5 |  |
| Tests unitaires | JUnit 3.8.1, DbUnit 2.1, JMOCK 1.1.0 |  |
| Gestion des transactions | Spring 2.0 |  |
| Concurrence d’accès | Hibernate 3.2 |  |

# Outils

## Intégration Continue

### Rappel de l 'utilité de l'intégration continue

* responsabilise le développeur,
* discipline le développeur,
* encourage l'écriture de tests automatisés,
* apporte une amélioration continue du code.

### Objectif de l'intégration continue

* Constituer et standardiser une plate-forme d'intégration continue.
* Mise en place d’un « Site Qualité » pour chaque projet JEE Mediapost:
  + Assurer la vérification et centraliser des bonnes pratiques de développement
  + Communication des résultats de l’équipe projet
  + Mise à disposition d’une synthèse des projets pour analyse globale
* Principes généraux de la plateforme
  + Exécution régulière des contrôles qualités du projet : Build Server – Continuum
    - Détecter au plus tôt les anomalies et anticiper le niveau de qualité final
  + Publication d’un site par projet sur un serveur web Apache accessible à tous
    - Un répertoire alloué à chaque projet pour contenir le site du projet
* Proposer une structure standard du projet
  + Un POM Parent dont chaque projet peut hériter – avec la génération d'un site et des rapports standards : voir [le POM Parent Mediapost](#scroll-bookmark-36) et [l'héritage de POM](#scroll-bookmark-37)
  + Des archetypes pour chaque type de projets (métier, JSF). le POM du prototype généré à partir d'un archetype hérite du POM Parent Mediapost : voir [Les archetypes Mediapost](#scroll-bookmark-38)

Les tests unitaires nous assurent que chaque classe est valide. Néanmoins, un certain nombre de points doivent être vérifiés :

* Les outils de « versionning » comme CVS ne permettent pas de s’assurer que tous les conflits sont, du point de vue fonctionnel, résolus
* Les développeurs peuvent être distraits lorsqu’ils effectuent les « commit » dans un CVS (Oublie de commit sur une classe modifiée)
* Les interfaces peuvent être modifiées. Le respect des tests unitaires permet de vérifier que les implémentations de l’interface sont valides, mais pas les classes utilisatrices de l’interface (même si cette erreur est détectable à la compilation).
* Des modifications sur les ressources peuvent êtres effectuées sans modifications des classes les utilisant (Modification du schéma de base de données, …)

Le premier rôle de l’intégration continue est de permettre, en lançant la compilation de tous les sources et de l’exécution de l’ensemble des tests, de vérifier que le projet est dans un état stable, à tout instant. Mais l’exécution automatique de la compilation et de l’ensemble des tests n’est pas suffisante. Il est nécessaire de s’assurer que nous sommes à tout instant capables de livrer une application stable sur l’environnement cible de production (ou le plus proche possible de l’environnement de production). Nous devons nous assurer que :

* Les fichiers d’initialisation sont toujours à jour
* Les fichiers de configuration sont toujours à jour
* Les développements sont compatibles avec l’environnement cible

### Les étapes de l'intégration continue

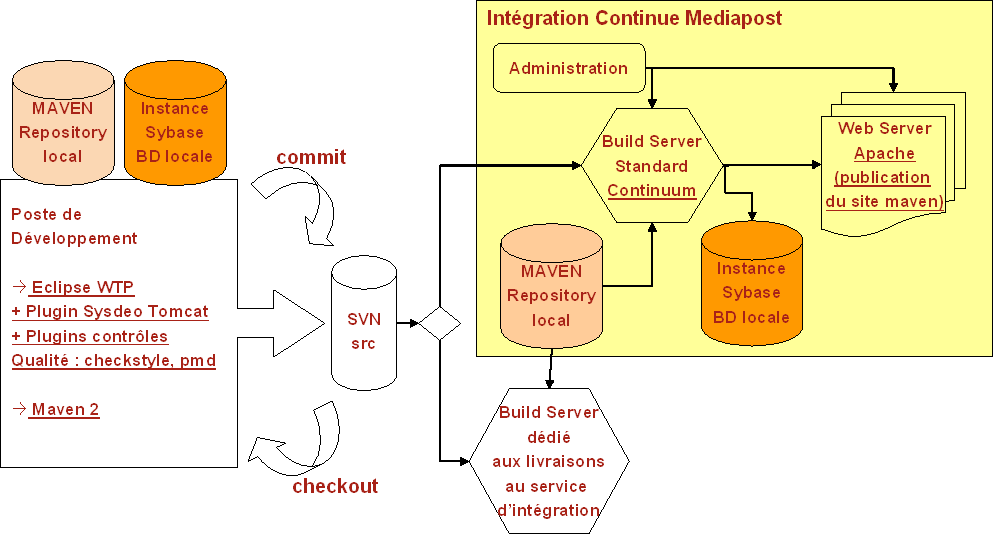
Les principales étapes de l’intégration continue sont:

* Récupération des sources du HEAD via le Système de contrôle de version (la cible est Subversion chez Mediapost)
* Compilation des sources (+Javadoc)
* Lancement des tests unitaires et constitution des rapports de tests
* Création de paquetage (binaires) SNAPSHOT (attention, ce n'est pas le processus d'intégration qui se charge de faire "releases", ceci est une opération manuelle)
* Vérification de la qualité de cette distribution (Cobertura, PMD, checkstyle, JDepend, JavaNCSS)
* Diffusion des rapports de l’intégration continue (génération du site)

### Une architecture technique dédiée à l'intégration continue

Pré-requis pour le serveur d'intégration continue:

* On suppose que tous les matins les développeurs font un checkout des sources et tous les soirs un checkout + une résolution éventuelle des conflits + un commit de leurs sources. Le repository SVN ou CVS est ainsi régulièrement mis à jour.
* Une base de données Sybase doit être disponible pour le processus d'intégration continue
* Maven est installé sur l'environnement où se déroule le processus d'intégration continue : un maven repository local et un fichier settings.xml existent pour cet environnement



### Les Outils

Les outils privilégiés sont Ant et Maven (et les plugins associés) pour gérer le processus de construction et des contôles qualité :

- Script Ant : il faut écrire toute les phases sous forme de target et l’utilisation de tasks spécialisées (cvs, junit, junitreport): long mais bon moyen pour maîtriser de bout en bout le processus.

- Maven : [http://maven.apache.org](http://maven.apache.org/) : préférable. Maven peut permettre d'automatiser ces différentes tâches. Voir [Maven](#scroll-bookmark-37)

**Nous avons choisi Maven 2 pour les projets Mediapost.**

Il existe des outils front-end spécifiques permettant de gérer le processus d’intégration continue (ant restant l’utilitaire central sous-jacent de la plupart de ces outils). La liste suivante n’est pas exhaustive mais permet de référencer quelques uns de ces outils :

* CruiseControl : [http://cruisecontrol.sourceforge.net](http://cruisecontrol.sourceforge.net/)
* Continuum : <http://maven.apache.org/continuum/>
* Luntbuild : <http://www.luntsys.com/luntbuid>
* Damage control : [http://damagecontrol.codehaus.org](http://damagecontrol.codehaus.org/)
* Anthill : <http://www.urbancode.com/projects/anthill/default.jsp>
* Gump : [http://gump.apache.org](http://gump.apache.org/)

**Pour avoir un comparatif exhaustif et tenu à jour** : cf. la matrice des outils d'intégration continue à <http://damagecontrol.codehaus.org/Continuous+Integration+Server+Feature+Matrix>

**Nous avons choisi Continuum pour la mise en place de l'intégration continue chez Mediapost.**

#### Présentation générale de Continuum

* **objectif de continuum**par rapport à d'autres tels que cruisecontrol :
  + simplicité et intégration parfaite avec maven2.
  + même niveau de fonctionnalité actuel que cruisecontrol.
* l'avenir avec la release 1.1 qui n'a pas de date de sortie prévue pour le moment :
  + notion de profils de builds
  + notion de groupe de projets pour les builders tous en même temps par exemple
  + le build d'un projet entraînera le build des projets qui dépendent de lui pour s'assurer la non régression
  + la possibilité de faire des release avec continuum

#### Installer et Relancer continuum

* assurez-vous que la variable d'environnement JAVA\_HOME est défini dans votre environnement
* télécharger la dernière version de Continuum : <http://maven.apache.org/continuum/download.html>. La version utilisée pour la première instalation de Continuum sur poste Windows est la 1.0.3
* l'installation de continuum est extrêmement simple : il suffit de dézipper l'archive téléchargée précédemment
  + sous Windows 2000/XP : dézipper cette archive dans le répertoire d'installation de Continuum (par exemple : C:\Program Files\Apache Software Foundation\continuum-1.x.x), soit CONTINUUM\_HOME ce répertoire
  + sous Unix (Linux, Solaris, etc.) : détarer cette archive dans le répertoire d'installation de Continuum (par exemple : /usr/local/continuum-1.x.x), soit *CONTINUUM\_HOME* ce répertoire
* exécuter le script suivant en ligne de commande:
  + *CONTINUUM\_HOME*/bin/linux/run.sh sous LINUX
  + *CONTINUUM\_HOME*/bin/solaris/run.sh sous SOLARIS
  + *CONTINUUM\_HOME*/bin/win32/run.bat sous Windows

L'installation peut durer plusieurs minutes (Continuum crée une base de données Derby et d'autres éléments d'initialisation). A la fin de cette tâche, vous devez voir apparaître la ligne suivante dans votre fenêtre de commande:

[INFO] The application server has started.

L'installation est alors terminée!

**A noter que le même script est à utiliser pour relancer Continuum.**

Lorsque Continuum est lancé, vous pouvez alors accéder via votre navigateur à l'url <http://localhost:8080/continuum/> pour configurer Continuum

#### Configurer continuum

Au premier lancement de la console <http://localhost:8080/continuum>, vous devez paramétrer différents élements. Le premier écran de configuration est du type :



* Spécifiez le login et le mot de passe d'un administrateur continuum
* Vous pouvez laisser les éléments par défaut pour les indications de configuration spécifiant :
  + le working directory : répertoire de travail de Continuum
  + le build-output directory : répertoire de sortie des livrables produits par le processus d'intégration continue
  + le deployment repository directory : (optionnel) répertoire de déploiement des livrables produits par le processus d'intégration continue

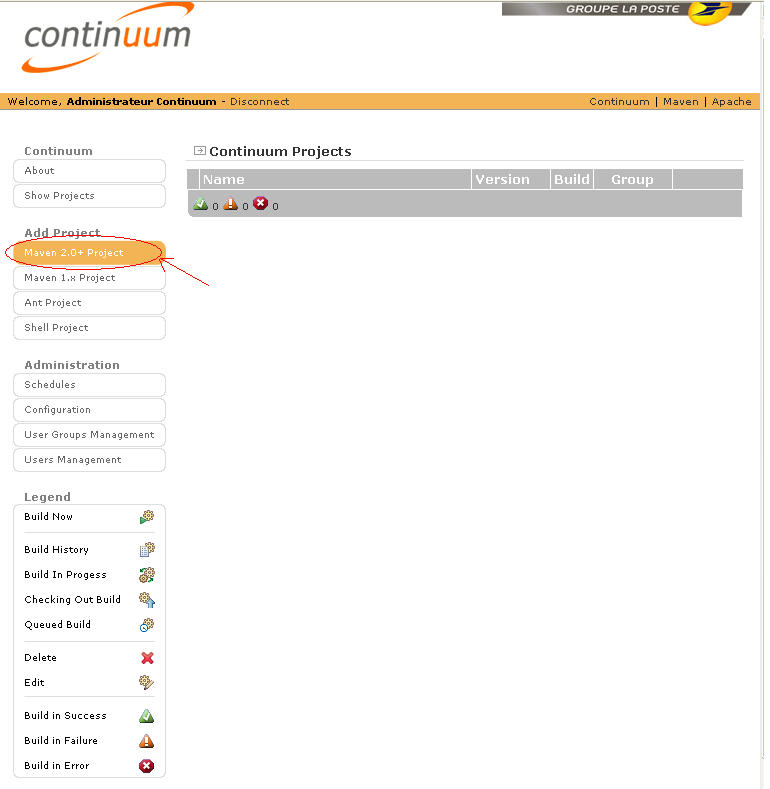
Une fois ces informations validée, la configuration est terminée. S'affiche alors l'écran ci-dessous, vous pouvez vous logger en indiquant le login et le mot de passe de l'administrateur continuum saisi lors de l'étape de configuration.



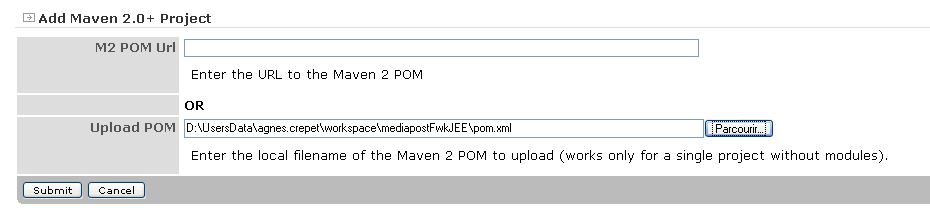
#### Administrer continuum

##### Ajouter un projet

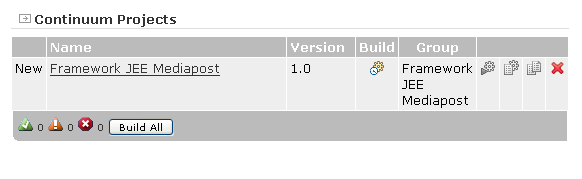
Une fois l'administrateur connecté à la plateforme, l'écran ci-dessous apparaît. Cliquer sur 'Add Project -> Maven 2.0 Project':



Ajouter un projet revient simplement à uploader le pom.xml du projet sur lequel on veut définir un processus d'intégration continue. Dans l'exemple ci-dessous, on ajoute le pom.xml du projet du framework Mediapost



Une fois le pom.xml du projet uploadé, l'écran ci-dessous aparaît. Vous pouvez alors définir les attributs du processus d'intégration continue sur ce projet:



##### Comment le processus d'intégration continue va récupérer les sources?

Attention, la première étape de l'intégration continue est la récupération des sources dans SubVersion ou CVS. Les premiers tests ont été réalisés avec CVS, nous parlerons de CVS ainsi dans la suite des explications mais les principes sont les mêmes quel que soit le système de contrôle de version : SVN ou CVS.

Il faut donc que le processus d'intégration continue recupère ces sources : on doit donc disposer dans l'environnement d'intégration continue d'un client CVS (Tortoise CVS par exemple). Attention, il faut bien ajouter à la variable PATH de votre environnement le répertoire du binaire cvs (C:\Program Files\TortoiseCVS par exemple).

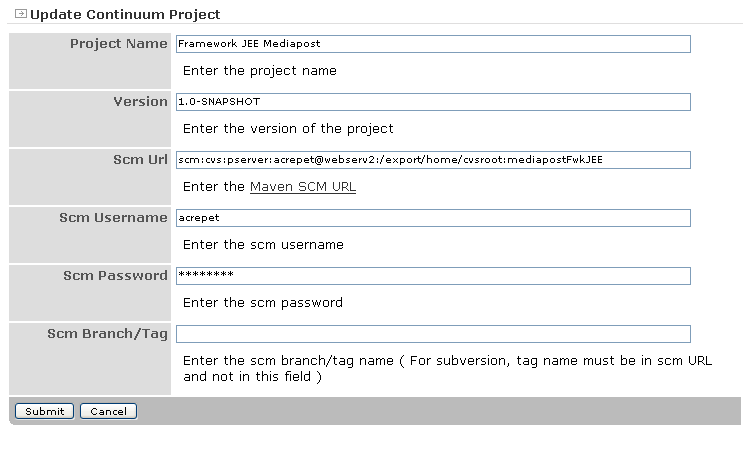
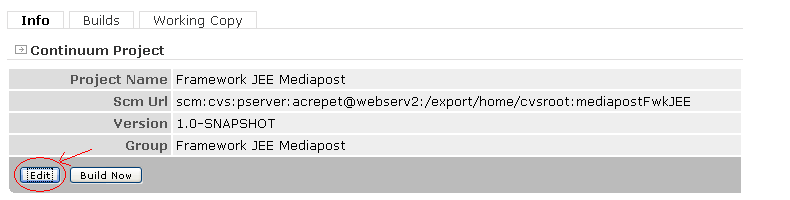
Un compte CVS pour l'administrateur continuum doit être créé (en lecture seule de préférence), soit admincontinnum ce compte. A l'installation du client CVS, un fichier .cvspass est créé à la racine du *Home Directory* de l'utilisateur qui lance ce processus d'intégration continue (utilisateur dans le sens utilisateur du Système d'Exploitation, par exemple D:\UsersData\agnes.crepet sous Windows).

Dans le pom.xml de chaque projet une section <scm>...</scm> précise les données de connexion au repository CVS : la sous-section <connection></connection> est utilisée par défaut par le processus d'intégration continue, elle est obligatoire. La sous-section <developerConnection></developerConnection> est utilisée pour créer les releases (voir [la création de release](http://sunic.mediapost.fr/dsiwiki/index.php?n=Maven.DeploiementDistribution#release)).

Extrait d'un fichier pom.xml d'un projet:

<scm>  
 <connection>  
 scm:cvs:pserver:admincontinnum@webserv2:/export/home/cvsroot:mediapostFwkJEE  
 </connection>  
 <developerConnection>  
 scm:cvs:pserver:acrepet@webserv2:/export/home/cvsroot:mediapostFwkJEE  
 </developerConnection>  
 </scm>

Dans continnum, il est possible de spécifier les données de connexion CVS ou SVN en cliquant sur le bouton 'Edit' de la section 'Continuum project' du projet:



##### Comment le processus d'intégration continue va se connecter à une base de données ?

La préconisation est que **le processus d'intégration continue n'encapsule pas lui-même le démarrage de la base de données**. Le démarrage de la base de données, tout comme la maintenance du schéma de la base de données est un pré-requis au bon déroulement du processus.

Il est nécessaire qu'une instance Sybase soit installée et démarrée sur l'environnement d'intégration continue (et éventuellement d'autres types de base de données, typiquement HSQLDB est utilisé dans les tests unitaires des DAOs du prototype MunicipostPos).

Néanmoins, un problème persiste : comment spécifier les bonnes datasources au processus d'intégration continue pour le déroulement des tests unitaires? En effet, le projet contient une configuration des datasource pour les environnements de développements (par exemple, le prototype MunicipostPos dispose de fichiers de configurations de datasources spécifiques dédiés aux tests unitaires des DAOs dans le fichier src/test.resources/datasource.properties).

Les datasources utilisées pour les tests unitaires sont donc externalisées dans un fichier de configuration : datasource.properties (voir prototype MunicipostPos). Extrait du fichier de configuration de Spring municipostPos-test-db.xml utilisé pour les tests unitaires des DAOs du prototype MunicipostPos:

<beans>  
...  
<bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" >  
<property name="location" value="classpath:datasource.properties"/>  
</bean>  
  
<bean id="dataSourceRs" destroy-method="close"  
 class="com.mchange.v2.c3p0.ComboPooledDataSource">  
 <property name="driverClass" value="${jdbc.driverClassName.rs}"/>  
 <property name="jdbcUrl" value="${jdbc.url.rs}"/>  
 <property name="user" value="${jdbc.username.rs}"/>  
 <property name="password" value="${jdbc.password.rs}"/>  
</bean>  
<bean id="dataSourceRc" destroy-method="close"  
 class="com.mchange.v2.c3p0.ComboPooledDataSource">  
 <property name="driverClass" value="${jdbc.driverClassName.rc}"/>  
 <property name="jdbcUrl" value="${jdbc.url.rc}"/>  
 <property name="user" value="${jdbc.username.rc}"/>  
 <property name="password" value="${jdbc.password.rc}"/>  
</bean>  
<bean id="dataSourceMa" destroy-method="close"  
 class="com.mchange.v2.c3p0.ComboPooledDataSource">  
 <property name="driverClass" value="${jdbc.driverClassName.ma}"/>  
 <property name="jdbcUrl" value="${jdbc.url.ma}"/>  
 <property name="user" value="${jdbc.username.ma}"/>  
 <property name="password" value="${jdbc.password.ma}"/>  
</bean>  
...  
</beans>

Ainsi un fichier datasource.properties doit être disponible dans le classpath du projet. Exemple de contenu de ce fichier utilisé pour les tests unitaires des DAOs du prototype MunicipostPos:

jdbc.driverClassName.ma=com.sybase.jdbc2.jdbc.SybDriver  
jdbc.url.ma=jdbc:sybase:Tds:recettemdp:4100/etudepna\_dev?JCONNECT\_VERSION=5  
jdbc.username.ma=sa  
jdbc.password.ma=pcsoft1  
  
jdbc.driverClassName.rc=com.sybase.jdbc2.jdbc.SybDriver  
jdbc.url.rc=jdbc:sybase:Tds:recettemdp:4100/refclient\_dev?JCONNECT\_VERSION=5  
jdbc.username.rc=sa  
jdbc.password.rc=pcsoft1  
  
jdbc.driverClassName.rs=com.sybase.jdbc2.jdbc.SybDriver  
jdbc.url.rs=jdbc:sybase:Tds:recettemdp:4100/refstruct\_odyss\_dev?JCONNECT\_VERSION=5  
jdbc.username.rs=sa  
jdbc.password.rs=pcsoft1

Il s'agit alors de surcharger ce fichier dans l'environnement d'intégration continue. Dans le pom.xml du rojet, il faut utiliser la configuration suivante (plugin maven resources):

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-resources-plugin</artifactId>  
 <executions>  
 <execution>  
 <id>test-resources</id>  
 <phase>process-test-resources</phase>  
 <goals><goal>resources</goal></goals>  
 <configuration>  
 <outputDirectory>  
 ${project.build.testOutputDirectory}  
 </outputDirectory>  
 <resources>  
 <resource>  
 <directory>${testResourcesDirectory}</directory>  
 <includes>  
 <include>datasource.properties</include>  
 </includes>  
 <filtering>false</filtering>  
 </resource>  
 </resources>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>

* ${project.build.testOutputDirectory} correspond au répertoire target/test-classes
* ${testResourcesDirectory} est à définir de manière spécifique au sein du fichier settings.xml maven (propre à chaque développeur). Ainsi une copie va être réalisée du fichier ${testResourcesDirectory}/datasource.properties vers le fichier target/test-classes/datasource.properties

##### Quel processus d'intégration continue va être lancé ?

Le processus d'intégration va lancer un goal maven. **Le pré-requis est donc que maven soit installé sur le serveur d'intégration continue.**

Rappelons que l'environnement dispose ainsi d'un fichier de configuration maven settings.xml. Voir [Installation de Maven](http://sunic.mediapost.fr/dsiwiki/index.php?n=Maven.PresentationGeneraleMaven#installation). Après l'installation de maven, ce fichier se trouve dans un répertoire .m2 à la racine du *Home Directory* de l'utilisateur qui lance le processus d'intégration continue (utilisateur dans le sens utilisateur du Système d'Exploitation). Exemple d'emplacement de fichier settings : D:\UsersData\agnes.crepet\.m2\setting.xml sous Windows.

**La préconisation est que le goal maven du processus d'intégration continue soit** site-deploy**.**

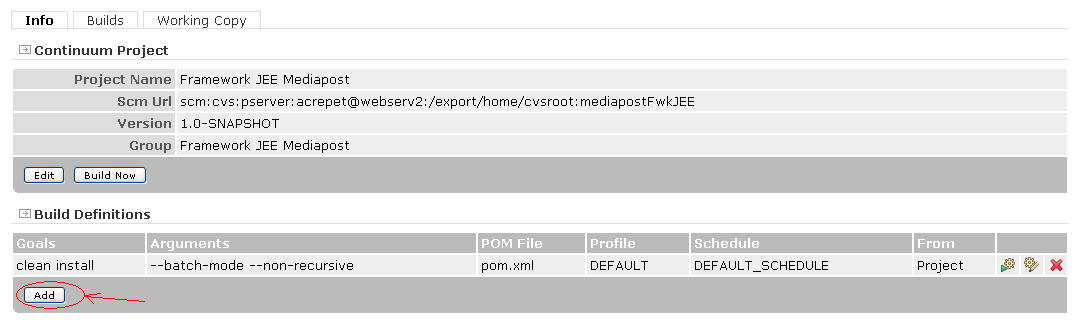
En effet la commande mvn site va généré un site maven (en lançant bien évidemment avant la compilation, les tests unitaires, la distribution, les contrôles qualité ) dans le répertoire de travail de continuum (sous le répertoire /target du projet). Or ce qui nous intéresse pour une mise en place optimale de l'intégration continue, c'est de pouvoir déployer le site sur un serveur Web (distant ou non) qui permettra à toute l'équipe de développement et même à d'autres équipes de Mediapost (équipe intégration par exemple) de visualiser les rapports maven. Pour déployer le site maven, non pas en local dans le répertoire de travail de Continuum, mais plutôt sur un serveur web, il faut spécifier dans le pom.xml la section suivante (pour des informations complémentaires, voir [la section sur le déploiement](http://sunic.mediapost.fr/dsiwiki/index.php?n=Maven.DeploiementDistribution)):

<distributionManagement>  
 <site>  
 <id>website Integration Continue</id>  
 <url>  
 ${deploy.site.root}/${project.artifactId}/${project.version}  
 </url>  
 </site>  
 </distributionManagement>

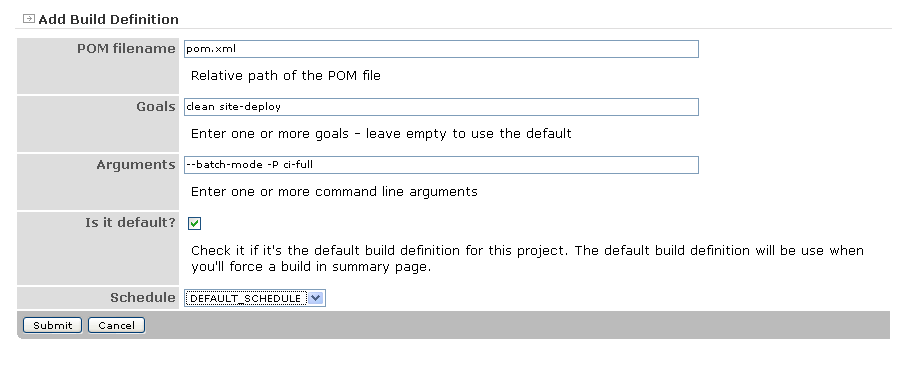
Il faut déclarer la propriété deploy.site.root dans le fichier de configuration maven settings.xml de l'environnement d'intégration continue. Nous préconisons de la faire pointer vers le répertoire de gestion des sites web d'Apache. Par exemple:

<settings>  
 ...  
 <activeProfiles>  
 ...  
 <activeProfile>user-properties</activeProfile>  
 </activeProfiles>  
 <profiles>  
 <profile>  
 <id>user-properties</id>  
 <properties>  
 <deploy.site.root>  
 file://localhost/C:/bin/apache/www/ci/sites  
 </deploy.site.root>  
 ...  
 </properties>  
 </profile>  
 </profiles>  
 </settings>

Pour paramétrer ce goal site-deploy, il faut définir un processus de *build* : à partir de l'écran d'administration du projet cible, cliquer sur le bouton 'Add' de la section **Build Definitions** :



L'écran suivant va s'afficher :

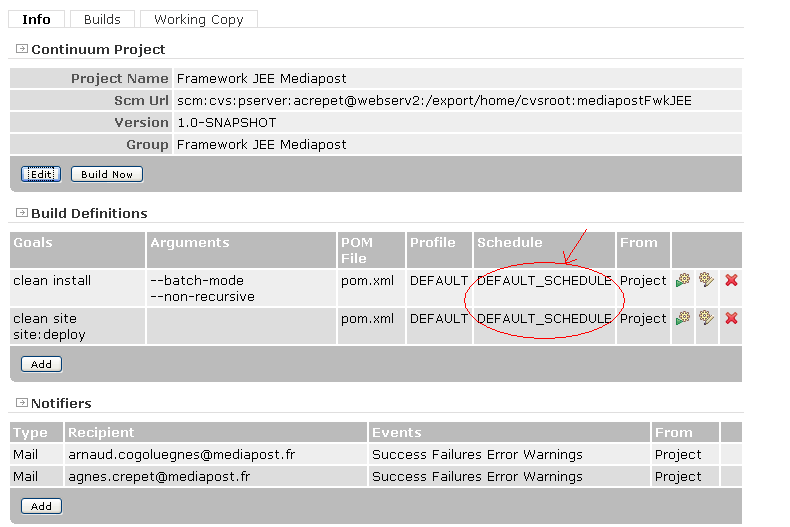


Vous allez pouvoir ainsi définir un nouveau goal comme ci-dessous. Il est important de noter :

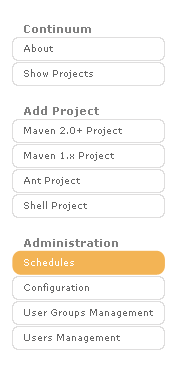
* qu'il faut lancer un goal clean avant le processus d'intégration lui-même pour nettoyer l'environnement
* que le lancement se fait en "batch", avec l'argument "--batch-mode", afin d'éviter que le processus ne se bloque si Maven demande d'entrer une valeur
* que le profil lancé est ci-full (profil complet avec tous les contrôles qualité) : voir [les profils Maven](http://sunic.mediapost.fr/dsiwiki/index.php?n=Maven.PresentationGeneraleMaven#profils)
* qu'il faut préciser qu'il s'agit du goal par défaut

##### Quand le processus d'intégration continue va t'il être lancé ?

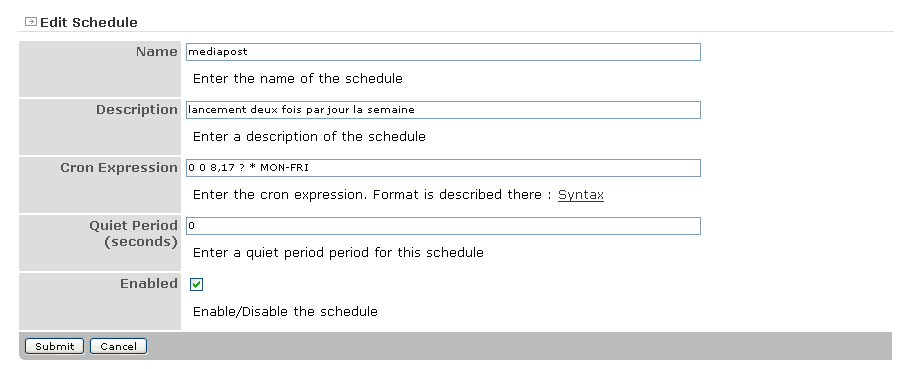
Chaque projet géré dans Continuum possède un SCHEDULE, c'est-à-dire une manière de lancer son processus d'intégration continue. Un SCHEDULE par défaut existe:



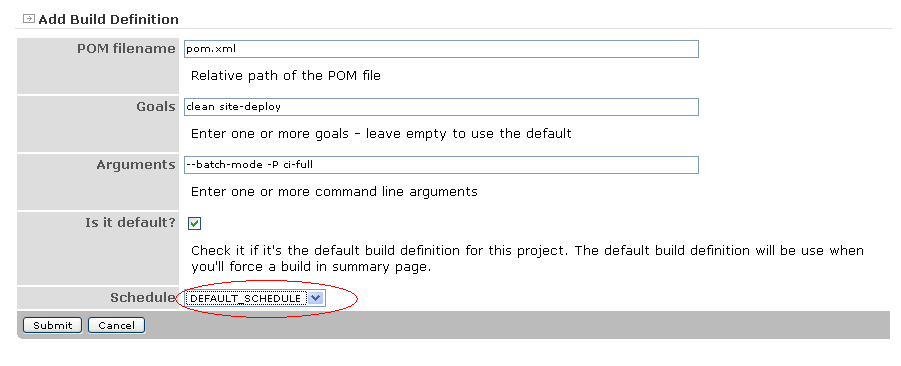
Il est possible de définir son propre SCHEDULE en cliquant dans le menu de gauche : Administration->Schedules:



**Nous préconisons de lancer le processus deux fois par jour, en fin et début de journée, et ce uniquement les jours de semaine et non le week-end. Voici la définition de ce schedule:**



Pour attribuer son propre SCHEDULE au projet, lors de définition d'un processus de build (section Build Definitions->Add) il faut choisir le bon SCHEDULE à partir de la liste déroulante:



### Liens vers des ressources complémentaires

* [<http://www.javaworld.com/javaworld/jw-01-2005/jw-0110-quality_p.html>] : Improve the Quality of your J2EE-Based projects : overview des différents éléments qui contribuent à la qualité de la production avec listing de certains applicatifs/plug-ins/anttask positionnés selon leur domaine de "compétence".
* [<http://java-source.net/open-source/code-coverage>] : Liste d'outils de CodeCoverage: un site qui propose un listing d'outils sur différents sujet, dont la couverture de code.
* [<http://www.mergere.com/m2book_download.jsp>] : Livre sur Maven 2.0: Mergere, une société fondée par les leaders de Maven, écrit un livre sur Maven 2.0 et sur l'intégration continue avec Continuum. Il est disponible gratuitement au download à partir de leur site.
* [<http://www.mergere.com/products.jsp>] : Mergere continue d'apporter des solutions packagées autour de l'intégration continue. Leur solution Maestro semble intéressante. A noter que dans Maestro, ils ont élaboré des scripts de migration ANT ou MAVEN vers MAVEN 2.

## Environnement de développement

### Distribution de Eclipse

La distribution de Eclipse préconisée est [Web Tools Platform](http://www.eclipse.org/webtools/). WTP comprend des plugins pour le développement J2EE (EJB, Web Service...) mais surtout de nombreux éditeurs pour tout type de fichiers, avec complétion. A ce jour les version les plus récentes sont :

* *1.5.1*, basée sur Eclipse 3.2
* *1.0.3*, basée sur Eclipse 3.1

La version 3.2 de Eclipse est stable, mais de nombreux plugin n'ont pas été portés dessus. Dès que l'on veut rajouter un plugin à la distribution WTP, il faut alors veiller à ce que celui-ci soit compatible avec Eclipse 3.2.

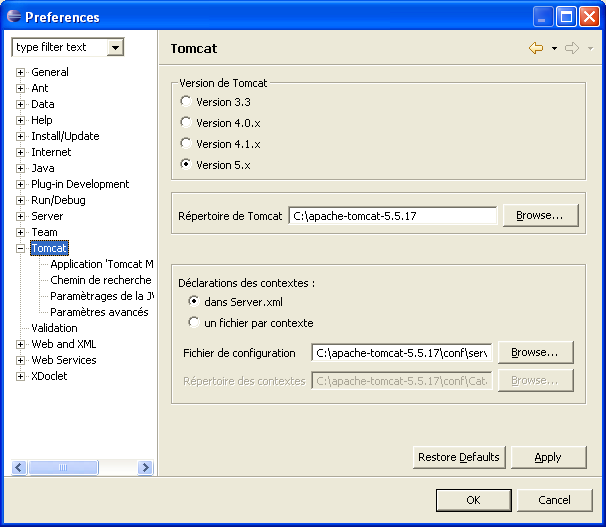
### Plugin Tomcat Sysdeo

Ce plugin permet de contrôler Tomcat à partir d'Eclipse. *Le plugin ne contient pas Tomcat, il faut récupérer une distribution du serveur séparément*. On peut vérifier que le plugin est installé correctement si la barre d'outils contient maintenant les icônes de Tomcat :

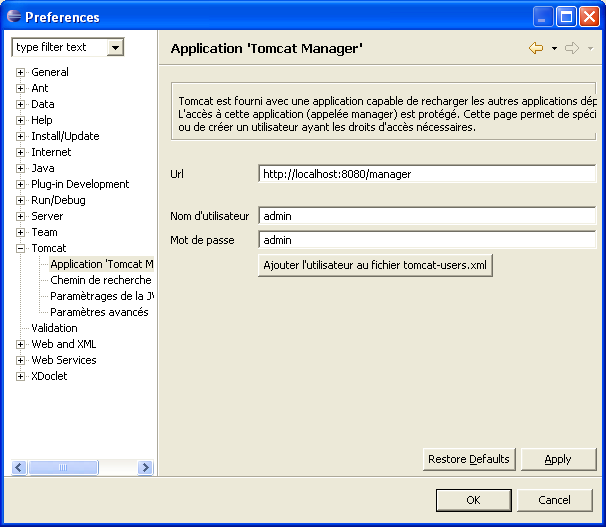


Ces icônes permettent (de gauche à droite) de lancer, arrêter et recharger Tomcat.

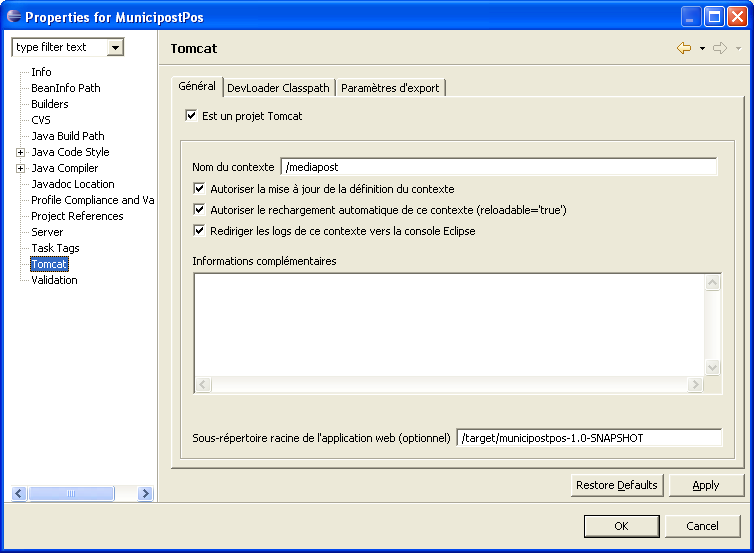
Une fois le plugin installé, on peut indiquer la localisation de Tomcat dans "Windows > Preferences > Tomcat" :



Il est aussi utile de préciser l'accès au manager de Tomcat pour pouvoir recharger les applications :



Si on veut qu'un projet utilise le plugin, il faut le référencer comme projet Tomcat, en cliquant bouton droit sur celui et en choisissant "Properties" puis "Tomcat" :



Il faut alors :

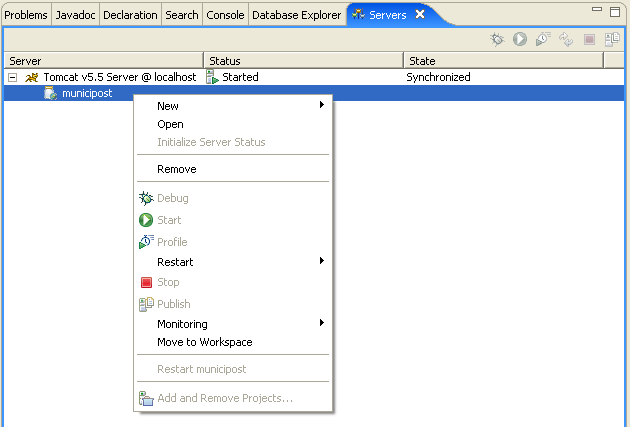
* cocher *Est un projet Tomcat*
* indiquer le nom du contexte, sans oublier l'anti-slash ('/')
* cocher les 3 cases en-dessous du contexte, afin notamment de rediriger la sortie de Tomcat vers la console d'Eclipse
* préciser où se trouve le répertoire de l'application Web, à partir de la racine du projet. En l'occurence ici ce sous-répertoire correspond au répertoire **target** utilisé par maven. En effet, lorsque vous lancez en ligne de commande mvn package, maven va alors compiler l'application et créer un répertoire **target** contenant l'image de la distribution de votre projet (qui permet alors de constituer l'archive .war ou .jar).

L'idée est donc d'utiliser maven pour créer votre distribution (mvn package), puis d'utiliser le plugin Eclipse Sysdeo pour Tomcat afin de tester votre application web. Ce plugin est fort commode pour développer plus rapidement vos applications web. Il permet notamment de debugger votre application, de rédeployer votre application sans devoir arreter Tomcat. L'alternative consistant à utiliser également maven pour déployer votre application sous le seveur tomcat de développement (mvn tomcat:deploy) n'est pas conseillée, elle s'avère beaucoup plus lourde à l'utilisation que le plugin sysdeo tomcat.

Une fois cela effectué, il suffit de lancer Tomcat avec l'icône correspondante.

### Alternative au plugin Tomcat

WTP inclue un plugin server qui permet de référencer des serveurs d'applications et notamment Tomcat. Une vue gère ces serveurs :



L'utilisation de cette vue peut être une alternative à l'utilisation du plugin. Son intégration à Eclipse est meilleure que celle du plugin Tomcat mais en contrepartie, le paramétrage est un peu plus complexe (car ce plugin permet d'utiliser n'importe quel serveur) et nécessite que le projet soit d'un type spécifique (*Dynamic Web Project*).

### Les autres plugin pour les contrôles qualité

Se reporter à la [description de ces outils et leur intégration dans Eclipse](http://sunic.mediapost.fr/dsiwiki/index.php?n=Outils.Qualite#QualiteEclipse)

## Builds avec Maven

Maven est un outil automatisant la construction, les tests, la documentation, le déploiement des applications java. Bien d’autres tâches disponibles. Maven = Ant++

Maven propose une gestion simplifiée des dépendances entre votre projet et les librairies annexes (open source ou interne)

Il assure le téléchargement à partir de référentiels Internet

Il offre un mécanisme de version des dépendances simplifié et très puissant

### Sommaire

* [Sommaire](#scroll-bookmark-57)
* [Installation](#scroll-bookmark-59)
* [Arborescence d'un projet Maven 2](#scroll-bookmark-60)
* [Rappel sur la gestion des dépendances](#scroll-bookmark-61)
* [Paramétrages](#scroll-bookmark-62)
  + [Que sont les propriétés système ?](#scroll-bookmark-63)
  + [Comment définir des variables système ?](#scroll-bookmark-64)
  + [Où trouver les références ?](#scroll-bookmark-65)
* [Pré-requis](#scroll-bookmark-66)
* [Tomcat](#scroll-bookmark-67)
* [Les tests](#scroll-bookmark-68)
* [Maven 2.1 : le contenu de la future version](#scroll-bookmark-69)
* [Maven Archetype ou la création de "blank project"](#scroll-bookmark-70)
* [Documenter son projet](#scroll-bookmark-71)
  + [Formats de sortie](#scroll-bookmark-72)
  + [Documentation](#scroll-bookmark-73)
  + [Génération du site](#scroll-bookmark-74)
  + [Déploiement du site](#scroll-bookmark-75)
* [Héritage de POM](#scroll-bookmark-76)
  + [Comment hériter d'un POM ?](#scroll-bookmark-77)
  + [Comment définir un POM "parent" ?](#scroll-bookmark-78)
* [Profils](#scroll-bookmark-79)
  + [Où peut-on définir les profils ?](#scroll-bookmark-80)
  + [Comment définir un profil ?](#scroll-bookmark-81)
  + [Comment activer un profil ?](#scroll-bookmark-82)
  + [Exemples avec des rapports](#scroll-bookmark-83)
  + [Ressources](#scroll-bookmark-84)

### Installation

* dézipper maven-2.0.4-bin.zip (vous pouvez récupérer l'archive dans le module CVS m2home)

dans un repertoire, soit C:\bin\maven ce repertoire.

* Positionner une variable d'environnement MAVEN\_HOME sur ce repertoire.

Pour créer une variable d'environnement : clic droit sur "Poste de Travail Windows"->Propriétés->Avancé->Variables d'environnement

* Mettre à jour la variable d'environnement PATH : lui ajouter %MAVEN\_HOME %\bin
* Créer votre repository maven local (répertoire où sont stockées toutes les librairies).

Nous avons créé un module m2home dans CVS. Celui-ci contient une image à jour du repository local maven 2 pour le prototype Mediapost, ainsi qu'un modèle du fichier settings.xml (fichier de configuration Maven). Chaque développeur peut donc disposer des éléments de ce module et procéder comme suit:

(%ECLIPSE\_HOME% correspond au répertoire d'installation d'Eclipse)

(Dans Window->Preferences->Java->Build Path->Classpath Variables). Sa valeur doit correspondre à votre repository local

la valeur de l'attribut <localRepository>.

Par exemple:

<localRepository>D:\UsersData\agnes.crepet\workspace\m2home\repo</localRepository>

* à la racine du projet (par exemple celui du prototype) vous pouvez lancer les goals en ligne de commande:
  + mvn clean (nettoyage de l'environnement)
  + mvn compile (compilation du projet)
  + mvn package (création du war)
  + mvn site (génération du site maven)
  + mvn tomcat:deploy (deploiement du .war sous Tomcat)

Voir le [Cycle de Vie de construction Maven 2](#scroll-bookmark-85)

### Arborescence d'un projet Maven 2

Un projet Maven est séparé classiquement en deux sous répertoires :

* src pour tout le code code source
* target pour les artefacts générés.

**Un artefact est le "livrable" de Maven** : un .jar, un .war, un site web, etc.

Les sources de l'application:

* src/main/java: code java de l'application
* src/main/resources: autres ressources requises par l'application
* src/main/filters: filtres des ressources (sous forme de fichiers properties)

qui peuvent être utilisés pour définir des variables seulement connues à l'exécution

* src/main/config: Fichier de configuration de l'application
* src/main/webapp: le répertoire Web de l'application pour le projet War
* src/test/java: tests unitaires
* src/test/resources: ressources utilisées pour les tests unitaires, mais qui ne sont pas déployées
* src/test/filters: filtres des ressources utilisées pour les tests unitaires, mais qui ne sont pas déployées
* src/site: fichiers utilisés pour générer le site web du projet maven

### Rappel sur la gestion des dépendances

Maven utilise un système très souple et assez intelligent pour la résolution des dépendances. L’utilisation la plus fréquente repose sur des dépendances identifiées par un numéro de version.

<dependency>

<groupId>springframework</groupId>   
 <artifactId>spring</artifactId>   
 <version>1.2.5</version>

</dependency>

Une utilisation plus judicieuse consiste à toujours utiliser la dernière version en date. Pour cela, il suffit d’encadrer le numéro de version par un crochet pour figer ou une parenthèse pour ouvrir le numéro de version. Par exemple, le numéro [1.2,) indique que la dépendance se fera sur le numéro 1.2.x ou supérieur. Etant donné la nature de certains projets, il n’est pas conseillé d’ouvrir en grand les numéros de version car on s’expose rapidement à des incompatibilité d’interfaces ; aussi est-il préférable d'utiliser [1.0,2.0) indiquant « toutes versions supérieure ou égale à 1.x et strictement inférieure à 2.x ».

<dependency>

<groupId>org.springframework</groupId>   
 <artifactId>spring</artifactId>   
 <version>[1.2,1.3)</version>

</dependency>

Dans cet exemple, on évite le risque d’incompatibilité des interfaces entre spring 1.2.x et spring 1.3.0.

### Paramétrages

Il existe plusieurs endroits dans Maven pour effectuer du paramétrage. L'idée consiste à avoir un paramétrage cohérent, qui assure une meilleure portabilité des builds :

* un ensemble de paramètres globaux, qui permettent de prendre en main l'environnement de développement rapidement
* des paramètres locaux, soit parce que ces paramètres sont en effet de portée locale (ex. : un chemin vers un fichier, un nom d'utilisateur...), soit parce qu'on désire, pour une raison ou une autre, surcharger les paramètres globaux

La configuration de Maven peut se faire dans 3 fichiers, du plus global au plus particulier :

* <MAVEN\_HOME>/conf/settings.xml (utilisé pour chaque lancement de Maven à partir du binaire <MAVEN\_HOME>)
* <USER\_HOME>/.m2/settings.xml (utilisé pour chaque lancement de l'utilisateur)
* dans le POM d'un projet (utilisé pour le projet)

Le fichier <MAVEN\_HOME>/conf/settings.xml est considéré comme le plus global quand Maven est installé en réseau, c'est-à-dire que tout le monde utilise les mêmes binaires. Si ce n'est pas le cas, c'est-à-dire que chaque développeur dispose d'une distribution de Maven sur son poste, <MAVEN\_HOME>/conf/settings.xml et <USER\_HOME>/.m2/settings.xml ont finalement la même portée, dans le sens où le développeur peut modifier les deux. Néanmoins, <USER\_HOME>/.m2/settings.xml aura le dernier mot pour un paramétrage, installation réseau ou pas.

Dans le cas d'une installation locale, il peut être judicieux d'avoir tout de même un <MAVEN\_HOME>/conf/settings.xml identique pour tous les postes. En revanche, chacun doit avoir sa propre version <USER\_HOME>/.m2/settings.xml, y compris le serveur d'intégration continue.

Voici des exemples de ce qui peut être réglé dans chacun des fichiers :

* <MAVEN\_HOME>/conf/settings.xml
  + les proxies, nécessaires à Maven pour accéder aux dépôts centraux
  + des profils, qui définissent eux-mêmes :
    - des dépôts internes, utilisés pour les distributions internes (librairies et plugins) de l'entreprise
    - des propriétés systèmes globales
  + l'activation par défaut de ces profils, voire de profils définis dans <USER\_HOME>/.m2/settings.xml
* <USER\_HOME>/.m2/settings.xml
  + des paramètres locaux (login, mot de passe) pour des servers
  + des profils, activés par défaut dans <MAVEN\_HOME>/conf/settings.xml et qui définissent alors des propriétés système locales
* dans le POM d'un projet
  + l'accès au dépôt de sources
  + les informations pour le déploiement, utilisant le plus possible des paramètres définis dans les 2 fichiers settings.xml

#### Que sont les propriétés système ?

Cette dénomination vient du monde Java, puisqu'il s'agit de valeurs accessibles par l'instruction System.getProperty("my.key"). Toujours en Java, il existe des propriétés système par défaut et on peut les définir en lançant le programme avec l'argument -D, par exemple -Dma.propriete=maValeur.

Avec Maven, on peut appeler ces propriétés dans les fichiers de configuration (settings.xml et pom.xml) avec la syntaxe ${ma.propriete}. Il existe des propriétés pré-définies : project.artifactId, project.version... Voici un exemple dans un fichier pom.xml :

<distributionManagement>   
 <site>  
 <id>website</id>  
 <url>  
 ${deploy.site.root}/${project.artifactId}/${project.version}  
 </url>  
 </site>  
</distributionManagement>

Dans cet exemple, l'URL pour le déploiement du site généré est définie avec 3 variables. Les variables project.artifactId et project.version sont propres au projet en cours et sont définies par Maven. En revanche, la variable deploy.site.root n'est pas une variable "standard" : elle a été définie quelque part, par le développeur (voir plus bas).

#### Comment définir des variables système ?

Les variables systèmes peuvent être définies :

* dans les fichiers settings.xml (pour un profil) :

<profiles>  
 <profile>  
 <id>global-properties</id>   
 <properties>  
 <ma.propriete>ma valeur</ma.propriete>  
 </properties>  
 </profile>  
</profiles>

* au lancement d'une commande Maven :

mvn site -Dmaven.test.skip=true

#### Où trouver les références ?

[Référence du POM](http://maven.apache.org/ref/2.0.3-SNAPSHOT/maven-model/maven.html)

[Référence de settings.xml](http://maven.apache.org/maven-settings/settings.html)

### Pré-requis

L’inconvénient majeur de Maven c’est qu’il nécessite un accès à internet sur tous les postes qui l’utilise.

Au-delà du fait que cela génère un important trafic sortant ce qui peut être pénalisant pour l’entreprise, cet accès n’est pas toujours ouvert à tous.

Néanmoins, lorsque le repository local sur le poste de développeur est complet et à jour, Maven ne va pas, à chaque déroulement de processus, récupérer et downloader sur le réseau les librairies.

A noter également qu'il est possible de configurer un proxy si nécessaire.

Pour cela, il s'agit de modifier le fichier %MAVEN\_HOME%\conf\settings.xml et d'ajouter une balise <proxies/>.

La configuration suivante a été mise en place pour Mediapost.

Extrait du fichier MAVEN\_HOME\conf\settings.xml:

<proxies>  
 <proxy>  
 <active>true</active>  
 <protocol>http</protocol>  
 <host>172.18.0.221</host>  
 <port>81</port>  
 <username/>  
 <password/>  
 <nonProxyHosts/>  
 </proxy>  
 </proxies>

Dans certains cas, Maven peut nécessiter une configuration très spécifique des fichiers projets (pom.xml) lors de l’utilisation de certaines librairies inaccessibles sur les repository généraux comme Ibiblio.

Rappelons juste que les outils de gestion de sources ne doivent pas être utilisés comme serveur de sauvegarde et que ce n’est pas une bonne pratique d’y stocker les librairies dont notre projet dépend. Ces librairies spécifiques doivent donc être déployées en local, sur chaque repository de chaque développeur ainsi que sur le serveur d’intégration continue.

Il peut parfois être utile d'installer dans le repository local une version d'un framework qui n'est pas encore postée sur le site de référence tel que ibiblio.

Typiquement, nous avons installé une version RC2 de Spring qui n'était encore à disposition sur les repository maven publiques.

Pour effectuer cette installation manuelle de librairie, il est nécessaire d'utiliser le plugin maven install.

Voici un exemple de ligne de commande pour installer Spring 2.0-RC3:

mvn install:install-file -DgroupId=org.springframework -DartifactId=spring \\  
-Dfile="D:\UsersData\agnes.crepet\Bureau\downloads\spring-framework-2.0-rc3\dist\spring.jar" -Dpackaging=jar -Dversion=2.0-RC3

Autre exemple (installation du plugin tomcat pour maven):

mvn install:install-file -DgroupId=org.codehaus.mojo\\  
-DartifactId=tomcat-maven-plugin-Dfile="D:\UsersData\agnes.crepet\Bureau\downloads\tomcat-maven-plugin-1.0-SNAPSHOT.jar"\\  
-Dpackaging=jar -Dversion=1.0-SNAPSHOT

### Tomcat

Pour déployer les archetypes .war, utilisez le maven plugin tomcat.

En pré-requis, vous devez avoir une variable d'environnement TOMCAT\_HOME positionnée.

Elle doit pointer vers votre répertoire d'installation de tomcat.

Vous devez ensuite simplement configurer votre pom.xml en ajoutant:

<build>  
 </plugins>   
 <plugin>  
 <groupId>org.codehaus.mojo</groupId>  
 <artifactId>tomcat-maven-plugin</artifactId>  
 <configuration>  
 <server>myserver</server>  
 <url>http://localhost:8080/manager</url> </configuration>  
 <version>1.0-SNAPSHOT</version>  
 </plugin>  
 </plugins>  
 </build>

et ajouter les lignes suivantes dans le fichier settings.xml:

<servers>  
 <server>  
 <id>myserver</id>  
 <username>admin</username>  
 <password>admin</password>  
 </server>  
 </servers>

### Les tests

Même si ceci n'est pas conseillé, il est parfois utile de ne pas dérouler l'execution des tests unitaires. Pour se faire, positionner dans votre fichier settings.xml:

<profiles>  
 ...  
 <profile>  
 <id>test\_config</id>  
 <properties>  
 <maven.test.skip>true</maven.test.skip>  
 </properties>  
 </profile>  
 ...  
 </profiles>  
 ...  
 <activeProfiles>  
 <activeProfile>test\_config</activeProfile>  
 </activeProfiles>

**Apport de Maven 2**

* différenciation entre qualité active et passive :

avant de commiter ou bien après avoir commité.

* possibilité avec maven 2 de faire échouer le build si le niveau de

qualité requis n'est pas atteint (nombre d'erreur de checkstyle, pourcentage de couverture de code des tests unitaires...)

* possibilité avec maven 2 de s'assurer de la compatibilité binaire entre

deux builds successifs.

* la best practice recommandée est de définir un niveau de qualité à respecter

sous peine de faire échouer le build. Dans le cadre de projet existant, le conseil est de commencer par un niveau assez facile à atteindre et de relever le niveau au fur et à mesure des releases / itérations.

* le futur :
  + mise en place d'un dashboard de qualité sur les projets maven2 (principe similaire à Xradar) avec des relevés dans le temps du niveau de qualité (checkstyle, complexité,...) ainsi que la possibilité de croiser ces informations avec d'autres indicateurs pour obtenir des métriques complexes telles que le ratio : pourcentage de test unitaire / nombre de ligne de code, etc
  + mise en place d'un principe de jauge d'activité des personnes intervenant sur les projets afin de pouvoir savoir qui a fait quoi sur quels projets et quel est par exemple le pourcentage de contribution par rapport aux autres.

### Maven 2.1 : le contenu de la future version

* possibilité de mettre des tags et des catégories standards sur les projets afin de pouvoir les retrouver par la suite avec maven repository manager par exemple
* optimisations au niveau de la gestion des connexions réseau
* meilleur support du versionning de l'héritage de pom
* meilleur support des maven embedder (outils permettant d'intégrer maven) pour des plugins d'IDE par exemple
* support de java 5 pour les annotations pour la construction de plugins
* support simplifié de plugins scriptables.

### Maven Archetype ou la création de "blank project"

Le plugin archetype de Maven permet de créer des blank project, c'est-à-dire des projets prêts à l'emploi. La plupart des tutoriaux Maven commencent par la création d'un projet, avec la commande suivante :

mvn archetype:create -DgroupId=com.mycompany.app -DartifactId=my-app

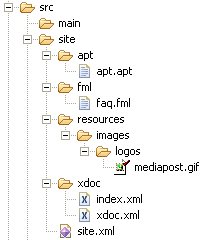
Cette commande permet de créer un projet *my-app* dans le groupe *com.mycompany.app*. Cela se concrétise par la création d'un répertoire qui contient toute l'arborescence Maven et un fichier pom.xml. Le projet est créé à partir d'un modèle par défaut et il est possible de créer ses propres modèles de projets, on parle alors d*'archetypes* pour évoquer ces modèles. Il est ensuite possible de crééer des projets "blank" qui sont alors aussi qualifiés de *prototypes*.

[Les archetypes Mediapost](#scroll-bookmark-38)

[Comment créer son archetype](#scroll-bookmark-86)

### Documenter son projet

Maven propose un système de documentation, qui permet, à partir de fichiers de formats divers, de documenter un projet. Dans l'arborescence Maven, le contenu de la documentation se trouve dans src/main/site. Une documentation minimaliste a l'apparence suivante :



Les éléments principaux sont :

* le plan du site, site.xml : on peut définir la bannière, le contenu du menu, des liens et l'emplacement des rapports générés (Checkstyle, tests unitaires, javadoc, sources...)
* une FAQ : on peut définir autant de FAQ que l'on veut, dans un format appelé *FML*
* des pages de contenu : elles peuvent être écrites en XDOC (format XML) ou en APT (Almost Plain Text, très proche du Wiki)
* des ressources (images)

A noter que le framework Mediapost contient une arborescence de ce type, avec ces principaux éléments. **Vous pouvez également utiliser [l'archetype documentation mediapost](#scroll-bookmark-87) pour générer cette arborescence. La création d'un prototype documentation à partir de cet archetype-documentation est lancée sur un projet existant : cette opération ajoute dans l'arborescence le répertoire site.**

Avec Maven, l'écriture de la documentation est orientée **contenu**, c'est-à-dire que l'on ne trouve pas ou très peu d'instruction de présentation dans les différents langages de documentation utilisés, mais plutôt des instructions sémantiques (<source>, <section>...). L'exemple typique est l'écriture de FAQ où l'on écrit une question et sa réponse à la suite, au format XML et le rendu se fait en affichant en premier lieu toutes les questions puis en deuxième lieu toutes les réponses.

#### Formats de sortie

L'intérêt du système de documentation orienté contenu est d'avoir plusieurs rendus possibles pour un même contenu. Maven 1 proposait principalement un rendu HTML et un rendu PDF. A l'heure d'aujourd'hui, Maven 2 propose seulement le rendu HTML, le PDF est en prévision.

#### Documentation

* [APT (Almost Plain Text)](http://maven.apache.org/guides/mini/guide-apt-format.html)
* [XDOC](http://maven.apache.org/maven-1.x/using/site.html) (documentation Maven 1)

#### Génération du site

Il suffit de lancer la commande mvn site. Le site est alors généré dans la target/site.

#### Déploiement du site

Il faut lancer la commande mvn site-deploy. La localisation du déploiement est indiquée dans le POM :

<distributionManagement>  
 <site>  
 <id>website</id>  
 <url>file://localhost/c:/Temp/sites/monprojet/${project.version}</url>  
 </site>  
</distributionManagement>

### Héritage de POM

Maven permet de définir de l'héritage entre POM, c'est-à-dire que les fichiers pom.xml de différents projets peuvent tous hériter de caractéristiques communes d'un même POM. Cela permet de partager les informations de l'entreprise, des dépendances, des instructions de déploiement, des rapports, etc cela afin de centraliser ses informations et aussi d'alléger les POM fils. **Un POM parent a été défini pour Mediapost : voir [Pom Parent Mediapost](#scroll-bookmark-36)**

#### Comment hériter d'un POM ?

Si on veut faire hériter un POM d'un POM parent, il faut ajouter le code suivant dans la balise project :

<parent>  
 <groupId>fr.mediapost</groupId>  
 <artifactId>mediapost-core</artifactId>  
 <version>1.0</version>  
</parent>

On remarque que l'héritage se fait en terme de plugin Maven : on indique le groupe et l'artifact (identifiant) du POM dont on veut hériter. Cela signifie donc que le dépôt Maven doit contenir ce plugin (d'une nature particulière car il n'est constitué que d'un POM).

#### Comment définir un POM "parent" ?

Un POM parent se définit un peu comme un projet Maven et plus particulièrement comme un plugin. Il suffit de créer un fichier pom.xml dans un répertoire et le remplir avec des informations le caractérisant (groupe, artifact...) et les informations qui sont destinées à être partagées avec ses fils (dépendances, rapports...). Voici par exemple un POM parent minimaliste :

<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">  
  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>fr.mediapost</groupId>  
 <artifactId>mediapost-core</artifactId>  
 <packaging>pom</packaging>  
 <version>1.0</version>  
 <name>POM parent Mediapost</name>  
 <url>http://yatoo.mediapost.fr/yatoo/</url>  
  
 <!-- dépendances -->  
  
 <dependencies>  
  
 <!-- dependances minimales -->  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>3.8.1</version>  
 <scope>test</scope>  
 </dependency>  
  
 </dependencies>  
  
</project>

On définit d'abord les informations identifiant le POM et surtout la nature de ce projet avec <packaging>pom</packaging>. Cela permet à Maven de savoir de quelle façon il doit déployer le projet dans le dépôt quand on lance la commande mvn install.

On définit ensuite des éléments que l'on veut partager, en l'occurrence une dépendance envers JUnit, dans une portée *test*.

Pour pouvoir utiliser ce POM en tant que POM parent, il faut l'installer dans le dépôt en se plaçant dans le répertoire du projet et en lançant

mvn install

Puis pour en hériter avec un nouveau projet :

<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">  
 <parent>  
 <groupId>fr.mediapost</groupId>  
 <artifactId>mediapost-core</artifactId>  
 <version>1.0</version>  
 </parent>  
  
 ...  
  
</parent>

### Profils

Maven 2 introduit la notion de profil qui permet de préciser et/ou surcharger certaines options de configurations dans un build. L'idée consiste à définir des options pour un profil et de lancer un build en précisant que l'on veut utiliser ce profil.

#### Où peut-on définir les profils ?

Les profils peuvent être définis à 3 endroits :

* dans le fichier settings.xml (utilisateur ou global)
* dans le POM d'un projet
* dans un fichier profiles.xml à la racine du projet

Les options de configuration possible sont différentes selon les endroits. Elles sont par exemple très limitées dans settings.xml car elles sont susceptibles d'influencer un grand nombre de projets. **Nous préconisons donc de définir les profils dans le POM d'un projet. Nous avons implémenté un exemple de définition de profils dans le** pom.xml **du projet parent Mediapost** mediapost-core**: voir [Pom Parent Mediapost](#scroll-bookmark-36)**

#### Comment définir un profil ?

Il suffit d'ajouter les informations suivantes dans le fichier concerné (préconisation : dans le POM, voir exemple mediapost-core/pom.xml ) :

<profiles>   
 <profile>  
 <id>ci-full</id>  
 </profile>   
</profiles>

On définit ici un profil ci-full (Intégration Continue complet). A noter que l'identifiant n'est pas obligatoire, mais utile pour avoir une idée de la fonction du profil. On peut également écrire un profil ci-light (Intégration Continue léger). A noter que nous avons implémenté dans le pom parent mediapost mediapost-core/pom.xml trois profils : ci-full et ci-light ainsi qu'un profil developper

#### Comment activer un profil ?

*Il est important de noter que l'on peut activer plusieurs profils en même temps*. Cela permet par exemple de diviser les éléments de configurations de façons logiques.

On peut activer un profil de différentes manières :

* directement en ligne de commande : mvn site -P ci-full. A noter qu'il est possible de spécifier plusieurs profils : mvn site -P ci-full,developper.
* en précisant le profil actif dans le fichier settings.xml:

<settings>  
 ...  
 <activeProfiles>  
 <activeProfile>ci-full</activeProfile>  
 </activeProfiles>  
 ...  
</settings>

* selon certaines conditions, par exemple le fait que l'on utilise le JDK 1.4 :

<profiles>  
 <profile>  
 <activation>  
 <jdk>1.4</jdk>  
 </activation>  
 ...  
 </profile>  
</profiles>

* en précisant une propriété système lorsque le build est lancé :

<profiles>  
 <profile>  
 <activation>  
 <property>  
 <name>debug</name>  
 </property>  
 </activation>  
 ...  
 </profile>  
</profiles>

Le profil sera par exemple activé si on lance le build de la manière suivante : mvn site -Ddebug.

* en précisant une propriété système et sa valeur lorsque le build est lancé :

<profiles>  
 <profile>  
 <activation>  
 <property>  
 <name>debug</name>  
 <value>on</name>  
 </property>  
 </activation>  
 ...  
 </profile>  
</profiles>

Le profil sera par exemple activé si on lance le build de la manière suivante : mvn site -Ddebug=on.

Il est aussi possible d'activer systématiquement un profil :

* lors de sa définition dans settings.xml :

<activation>  
 <activeByDefault>true</activeByDefault>  
</activation>

* toujours dans settings.xml, dans la balise activeProfiles :

<activeProfiles>  
 <activeProfile>property-overrides</activeProfile>  
 <activeProfile>default-repositories</activeProfile>  
</activeProfiles>

#### Exemples avec des rapports

Il peut être intéressant de préciser les rapports à produire selon le profil : beaucoup de rapport lors du build nocturne, peu de rapport pour un build de développeur :

<profiles>  
 <!-- profil integration continue complet -->  
 <profile>  
 <id>ci-full</id>  
 <activation>  
 <property>  
 <name>ci-full</name>  
 </property>  
 </activation>  
 <reporting>  
 <plugins>  
 <!-- informations sur le projet -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-project-info-reports-plugin</artifactId>  
 </plugin>   
 <!-- génération des sources au format HTML -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-jxr-plugin</artifactId>  
 </plugin>  
 <!-- rapports tests unitaires -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-surefire-report-plugin</artifactId>  
 </plugin>  
 <!-- rapports d'outils d'audit de code -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-pmd-plugin</artifactId>  
 <configuration>  
 <!-- les sources sont en 1.5 -->  
 <targetJdk>1.5</targetJdk>  
 </configuration>  
 </plugin>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-checkstyle-plugin</artifactId>  
 </plugin>  
 <plugin>  
 <groupId>org.codehaus.mojo</groupId>  
 <artifactId>cobertura-maven-plugin</artifactId>   
 </plugin>   
 <plugin>  
 <groupId>org.codehaus.mojo</groupId>  
 <artifactId>javancss-maven-plugin</artifactId>  
 <version>2.0-beta-1</version>  
 </plugin>  
 <plugin>  
 <groupId>org.codehaus.mojo</groupId>  
 <artifactId>jdepend-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </reporting>  
 </profile>  
 <!-- profil developpeur -->  
 <profile>  
 <id>developper</id>  
 <activation>  
 <property>  
 <name>developper</name>  
 </property>  
 </activation>  
 <reporting>  
 <plugins>   
 <!-- génération des sources au format HTML -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-jxr-plugin</artifactId>  
 </plugin>  
 <!-- rapports tests unitaires -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-surefire-report-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </reporting>  
 </profil>  
</profiles>

Dans cet exemple, le profil ci-full dispose de beaucoup de rapports tandis que le profil developper dispose du minimum.

La génération du site peut se lancer pour le profil ci-full :

mvn site -Dci-full

ou

mvn site -P ci-full

et pour le profil developper :

mvn site -Ddevelopper

ou

mvn site -P developper

#### Ressources

[Mini-guide sur les profils Maven 2](http://maven.apache.org/guides/introduction/introduction-to-profiles.html)

### Archétypes Mediapost

Afin d'accélérer et d'uniformiser la création de nouvelles applications, des archetypes ont été créés :

* un archetype pour une application JSF : jsf-archetype
* un archetype pour une bibliothèque métier : metier-archetype
* un archetype pour ajouter de la documentation à un projet Maven : [documentation-archetype](#scroll-bookmark-87)
* un archetype pour une application JSF-Facelets : facelets-archetype
* un archetype pour une application sécurisé par ACEGI : [acegi-archetype](#scroll-bookmark-90)

Pour créer un **prototype** (ce qui correspond à créer un projet basé sur un archetype), il faut lancer la commande suivante, on suppose dans cet exemple que l'on désire utiliser l'archetype metier-archetype :

mvn archetype:create -DarchetypeGroupId=fr.mediapost -DarchetypeArtifactId=metier-archetype -DarchetypeVersion=1.0-SNAPSHOT \\  
-DgroupId=fr.mediapost -DartifactId=kiosque

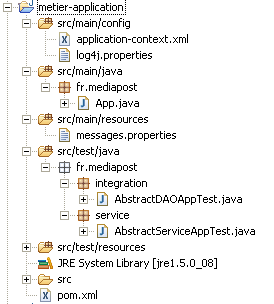
Attention, la ligne de commande de création du prototype ne doit pas être lancée à partir du répertoire du projet concerné (aucun répertoire n'est à créer en effet avant de lancer cette commande), mais à partir du répertoire parent (workspace Eclipse par exemple).

On distingue dans cette ligne de commande:

* le plugin à utiliser et le *goal* : archetype:create
* les paramètres de l'archetype à utiliser :
  + -DarchetypeGroupId=fr.mediapost (le groupe, qui ne change jamais dans notre cas)
  + -DarchetypeArtifactId=metier-archetype (l'identifiant de l'archetype)
  + -DarchetypeVersion=1.0-SNAPSHOT (la version de l'archetype)
* les paramètres du projet à créer :
  + -DgroupId=fr.mediapost (le groupe, qui ne change jamais dans notre cas)
  + -DartifactId=kiosque (le nom du projet)

#### metier-archetype , pour une bibliothèque métier

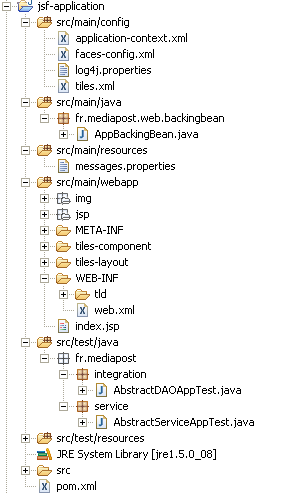
Un projet créé avec cet archetype a l'allure suivante :



Il constitue la base pour une bibliothèque métier, utilisant Spring et Hibernate.

#### jsf-archetype , pour une application JSF

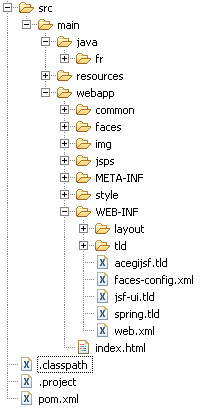
Un projet créé avec cet archétype a l'allure suivante :



Il constitue la base pour une application JSF, utilisant MyFaces, Spring et Hibernate. Il inclut metier-archetype

#### facelets-archetype , pour une application JSF avec Facelets

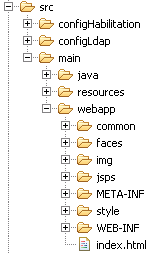
Un projet créé avec cet archétype a l'allure suivante :



Il constitue la base pour une application JSF, utilisant Facelets, MyFaces, Spring et Hibernate.

#### acegi-archetype , pour une application sécurisée par ACEGI

Un projet créé avec cet archétype a l'allure suivante :



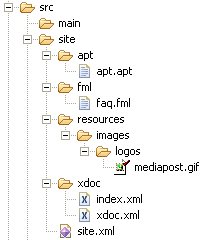
Il constitue une ébauche d'application sécurisée par ACEGI, utilisant ACEGI, Facelets, MyFaces, Spring et Hibernate.

mvn archetype:create   
 -DarchetypeGroupId=fr.mediapost   
 -DarchetypeArtifactId=acegi-archetype   
 -DarchetypeVersion=1.0-SNAPSHOT   
 -DgroupId=fr.mediapost   
 -DartifactId=demoprojet

Cet archétype permet également d’utiliser l’identification via NTLM.  
Pour cela il faut modifier le code généré dans la configuration de sécurité : context-acegi-security.xml. Il suffit commenter la partie dédié à l’identification par formulaire (rechercher FORM) et dé-commenter celle liée à NTLM.  
Attention, dans ce cas, les rôles ne seront plus ceux du fichier user.properties mais uniquement ceux issu du chargement normal (dans l’annuaire et la base de donnée cf. [Processus d’identification](http://sunic.mediapost.fr/dsiwiki/index.php?n=Transverse.Securite#identification)) .

#### documentation-archetype , pour créer de la documentation

Cet archétype a vocation à être appelé sur un projet existant. Il ajoute dans l'arborescence le répertoire site :

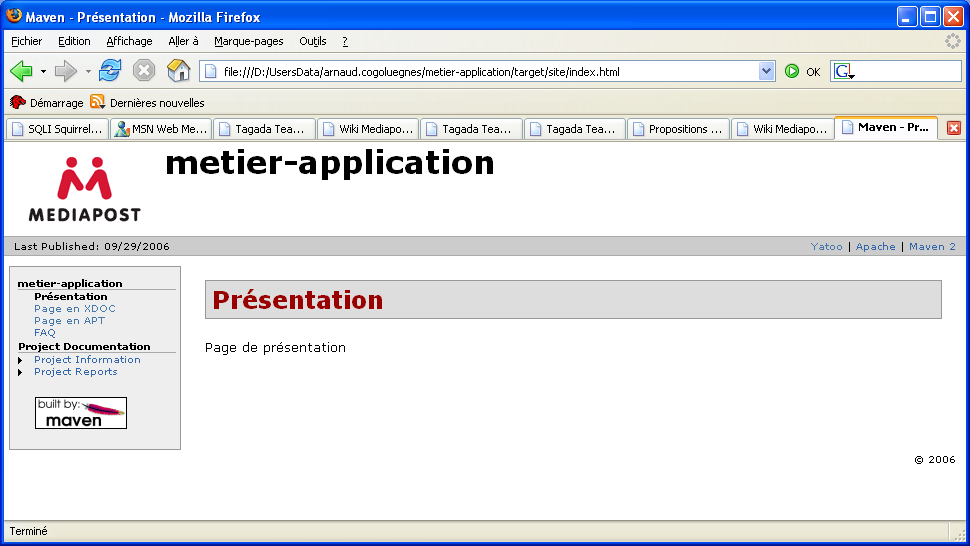


L'archétype contient l'ossature de la documentation :

* le plan du site (site.xml)
* une FAQ exemple
* un fichier APT
* des fichiers XDOC
* le logo Mediapost

Pour incorporer cet archétype à un projet, la procédure est la même que pour créer un prototype, *la ligne de commande de création du prototype ne doit pas être lancée à partir du répertoire du projet concerné, mais à partir du répertoire parent*.

La documentation est générée lors de la génération du site Web, en tapant la commande mvn site :



### Comment créer son archétype

Créer un archétype revient à écrire un plugin Maven assez simple. Un archétype est constitué des éléments suivants :

* un descripteur archetype.xml qui contient notamment l'identifiant de l'archétype et les fichiers qui le constituent
* les fichiers de ressources du prototype
* le fichier projet pom.xml du prototype (qui constitue en fait un fichier de ressource spécifique)
* le fichier projet pom.xml de l'archétype

**Nous avons créé cinq archétypes disponibles sous les modules cvs : [jsf-archetype](#scroll-bookmark-97), [metier-archetype](#scroll-bookmark-98) et [documentation-archetype](#scroll-bookmark-87), [facelets-archetype](#scroll-bookmark-99), [acegi-archetype](#scroll-bookmark-90) : voir la section dédiée aux [archetypes Mediapost](#scroll-bookmark-38)**

Il est important de noter que les fichiers de ressources sont en fait des templates Velocity, c'est-à-dire que l'on peut y insérer des variables qui seront remplacées lors de la création du prototype.

Voici une représentation de l'arborescence d'un archétype minimaliste :

archetype  
|-- pom.xml  
`-- src  
 `-- main  
 `-- resources  
 |-- META-INF  
 | `-- archetype.xml  
 `-- archetype-resources  
 |-- pom.xml  
 `-- src  
 |-- main  
 | `-- java  
 | `-- App.java  
 `-- test  
 `-- java  
 `-- AppTest.java

Voyons les éléments un par un.

#### pom.xml de l'archétype

Le fichier projet de l'archétype (qui est rappelons-le un plugin), a l'allure suivante :

<project>  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>fr.mediapost</groupId>  
 <artifactId>metier-archetype</artifactId>  
 <version>1.0</version>  
 <packaging>maven-plugin</packaging>  
</project>

On précise donc le groupe, l'identifiant et la version de l'archétype, ainsi que la version du POM et le packaging.

#### Le descripteur de l'archétype, archetype.xml

Ce fichier doit être dans le répertoire src/main/resources/META-INF/. Il doit préciser les différents fichiers que va contenir le prototype. Voici un exemple :

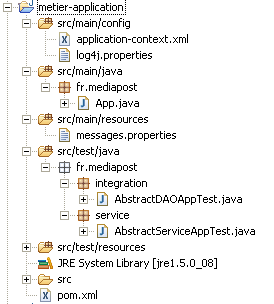
<archetype>  
 <id>metier-archetype</id>  
 <sources>  
 <source>src/main/java/App.java</source>  
 </sources>  
 <resources>  
 <resource>src/main/resources/messages.properties</resource>  
 <!-- configuration -->  
 <resource>src/main/config/log4j.properties</resource>  
 <resource>src/main/config/application-context.xml</resource>  
 <!-- projet Eclipse -->  
 <resource>.project</resource>  
 <resource>.classpath</resource>  
 <!-- site web maven -->  
 <resource>src/site/site.xml</resource>  
 <resource>src/site/resources/images/logolaposte3.png</resource>  
 </resources>  
 <testSources>  
 <source>src/test/java/AppTest.java</source>  
 </testSources>  
 <testResources>  
 <resource>src/test/resources/x.x</resource>  
 </testResources>  
</archetype>

On précise :

* l'identifiant de l'archétype avec <id>metier-archetype</id>
* les fichiers sources (Java) à inclure dans le projet avec <sources><source>src/main/java/App.java</source></sources>. A la création du prototype, les fichiers précisés se verront copiés dans l'arborescence correspondant au nom de package du groupe (dans notre exemple, App.java sera copié dans l'arborescence src/main/java/fr/mediapost)
* les fichiers de ressources sont précisés entre les balises <resources>...</resources> avec des éléments <resource>
* les fichiers Java des tests unitaires ainsi que leurs ressources (idem que pour les sources du projet)

NB : il n'est pour l'instant pas possible de préciser des répertoires vides

Le projet de cet archétype a donc l'allure suivante :



#### pom.xml du prototype

Il s'agit du fichier projet qui va être utilisé pour le prototype. Celui-ci n'a pas à être précisé dans archetype.xml. Voici un exemple :

<project>  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>${groupId}</groupId>  
 <artifactId>${artifactId}</artifactId>  
 <packaging>jar</packaging>  
 <version>${version}</version>  
 <name>Nom du projet</name>  
 <url>http://yatoo.mediapost.fr/yatoo/</url>  
</project>

On voit dans ce fichier des variables Velocity : ${groupId}, ${artifactId} et ${version}. Elles seront remplacées par les valeurs précisées par la commande archetype:create lors de la création du prototype.

Il est aussi possible de rajouter d'autres éléments au POM, par exemple les dépendances et les rapports désirés.

#### Templates Velocity pour les fichiers Java

Les fichiers précisés dans les balises sources ou testSources sont créés dans le prototype selon l'arborescence du groupe auquel ils appartiennent. Pour le groupe fr.mediapost (précisé lorsqu'on lance la commande la création du prototype archetype:create), les fichiers Java seront donc créés dans un répertoire fr/mediapost. Pour éviter d'avoir un problème à la compilation, il faut que les fichiers Java soient dans le package correspondant. Il suffit donc d'utiliser les variables Velocity :

package ${groupId};  
  
public class App {  
  
}

Le package de la classe et sa localisation seront alors en adéquation.

#### Installation et Mise à jour de l'archetype

Afin de rendre l'archétype utilisable, il faut le déployer et l'installer dans le dépôt Maven. Il suffit de lancer dans le répertoire de l'archétype (par exemple jsf-archetype ou metier-archetype) la commande :

mvn install

**Cette commande est à utiliser chaque fois que le contenu du projet de l'archetype a été modifié.**

Pour utiliser l'archetype et alors **créer un prototype**, il suffit de lancer la commande suivante à partir du répertoire parent du répertoire du projet que l'on veut créer (par exemple le workspace Eclipse):

mvn archetype:create -DarchetypeGroupId=<archetype-groupId> -DarchetypeArtifactId=<archetype-artifactId> \   
 -DarchetypeVersion=<archetype-version> -DgroupId=<my.groupid> -DartifactId=<my-artifactId>

les paramètres de l'archetype à utiliser :

les paramètres du projet à créer :

et le prototype sera créé dans le répertoire <my-artifactId> à partir du répertoire courant.

#### Mes images ne passent pas !

Il peut arriver que l'on veuille mettre une image dans un archétype, par exemple pour un archétype de documentation. Une fois le prototype créé, on s'aperçoit que l'image est illisible. Cela vient du fait que comme toutes ressources, l'image est considérée comme un template Velocity et que son passage dans le moteur de template a beaucoup de chances de l'altérer aléatoirement. *Cela est valable pour tout type de fichiers binaires*. Il faut donc indiquer dans le archetype.xml que l'image ne doit pas être considérée comme un template Velocity, en mettant l'attribut filtered à false :

<resource filtered="false">src/site/resources/images/logos/mediapost.gif</resource>

#### DTD de archetype.xml

Le fichier archetype.xml ne dispose pas de DTD mais voilà à quoi elle pourrait ressembler si elle existait (sources non-officielles) :

<!DOCTYPE archetype [  
 <!ELEMENT archetype ( id, allowPartial?, sources?, testSources?, resources?, testResources?, siteResources? )>  
 <!ELEMENT id (#PCDATA) >  
 <!ELEMENT allowPartial (#PCDATA) >  
 <!ELEMENT sources (source\*) >  
 <!ELEMENT testSources (source\*) >  
 <!ELEMENT resources (resource\*) >  
 <!ELEMENT testResources (resource\*) >  
 <!ELEMENT siteResources (resource\*) >  
  
 <!ATTLIST source encoding CDATA #IMPLIED >  
 <!ATTLIST resource encoding CDATA #IMPLIED >  
 <!ATTLIST resource filtered CDATA #IMPLIED >  
]>

### Cycles de vie de construction Maven

Maven 2.0 est basé sur le concept central de **cycle de vie de construction** (build lifecycle). Ainsi, le processus de construction et de définition d'un artefact particulier (c'est-à-dire le *livrable* de Maven : un .jar, un .war, etc.) est clairement défini.

La personne qui va utiliser Maven pour construire son projet devra donc connaitre un ensemble restreint de commandes pour construire un projet Maven, et le POM va garantir qu'elle obtienne le résultat attendu.

La phases centrales du cycle de vie de construction maven sont les suivantes :

* validate - valide que le projet est correct et que toutes les informations sont disponibles
* compile - compile le code source du projet
* test - teste le code source compilé en utilisant un framework de test adapté (JUNIT par exemple). Le passage de ces tests ne nécessite pas que le code soit packagé ou déployé.
* package - récupère le code source compilé et le package dans un format distribuable, comme un jar
* integration-test - lance et déploie le package si nécessaire dans un environnement d'intégration où des tests d'intégration peuvent être déroulés.
* verify - exécute tous les contrôles pour vérifier que le package soit valide et réponde aux critères de qualité
* install - installe le package dans le repository local, pour qu'il soit utilisé comme une dépendance pour un autre projet localement
* deploy - déploie le package dans un environnement d'intégration ou de production, et le copie dans le repository distant pour qu'il soit partagé par d'autres développeurs et projets

A noter que pour chacune de ces étapes, toutes les étapes précédentes sont toujours exécutées. Par exemple

mvn install

va lancer les étapes : compile , test, package, verify et installer le package dans le repository local

### Deploiement et distribution

#### Déploiement

##### Principe

Maven permet d'automatiser le déploiement : compilation, lancement des tests et *déploiement* de l'artefact sur un dépôt, la plupart distant, car l'artefact doit être mis à disposition. Pour cela, il faut *ajouter et renseigner une balise distributionManagement dans le POM du projet concerné* :

<distributionManagement>  
 <repository>  
 <id>mycompany-repository</id>  
 <name>MyCompany Repository</name>  
 <url>scp://repository.mycompany.com/repository/maven2</url>  
 </repository>  
</distributionManagement>

L'élément intéressant est l'url. Dans l'exemple, il fait référence à un serveur distant, auquel on accède via SSH2 (protocole scp). Il faut donc aussi fournir des éléments d'authentification, qui ne doivent pas apparaître dans le POM, car celui-ci est partagé par tous les développeurs du projet. Ces éléments étant propre à un utilisateur, ils doivent être logiquement définis dans USER\_HOME/.m2/settings.xml. Comment Maven sait-il ce qu'il faut récupérer ? Cela lui est indiqué par la balise <id>mycompany-repository</id>, qui fait référence à un serveur défini dans USER\_HOME/.m2/settings.xml :

<settings>  
 .  
 .  
 <servers>  
 <server>  
 <id>mycompany-repository</id>  
 <username>jvanzyl</username>  
 <!-- Default value is ~/.ssh/id\_dsa -->  
 <privateKey>/path/to/identity</privateKey> (default is ~/.ssh/id\_dsa)  
 <passphrase>my\_key\_passphrase</passphrase>  
 </server>  
 </servers>  
 .  
 .  
</settings>

Cet exemple est un peu complexe mais constitue une bonne illustration du principe de paramétrage de Maven, assurant une bonne portabilité des builds. Un déploiement peut être beaucoup plus simple s'il est fait en local :

<distributionManagement>  
 <repository>  
 <id>mycompany-repository</id>  
 <name>MyCompany Repository</name>  
 <url>file:/home/maven/repo/deploy</url>  
 </repository>  
</distributionManagement>

Le déploiement en local peut par exemple être utilisé pour les builds d'une plateforme d'intégration continue.

##### Types de déploiement

Les déploiements les plus fréquents concernent une release, un snapshot (développement) et les sites web générés. Voici comment faire la configuration dans le POM :

<distributionManagement>  
 <repository>  
 <id>release</id>  
 <url>file:/home/maven/repo/deploy</url>  
 </repository>  
 <snapshotRepository>  
 <id>snapshot</id>  
 <url>file:/home/maven/repo/deploy</url>  
 </snapshotRepository>  
 <site>  
 <id>website</id>  
 <url>file:/home/maven/monprojet/site</url>  
 </site>  
</distributionManagement>

Cet exemple montre les éléments minimum pour faire tous ces types de déploiements. Il est nécessaire de préciser la forme des déploiements afin de régler judicieusement les URLs.

###### Forme d'un déploiement release ou snapshot

Le déploiement d'une release ou d'un snapshot est destiné à un dépôt Maven. C'est-à-dire qu'il se sous l'arborescence REPOSITORY\_URL/groupId/artifactId/version/. Les fichiers produits sont notamment le JAR mais aussi le POM et des checksum.

Pour une release, le fichier JAR aura pour nom artifactId-version.jar. En revanche, pour un snapshot, comme le numéro de version est toujours le même et qu'un snapshot peut être déployé très souvent (tous les jours en général), le fichier JAR a la forme artifactId-version-date-heure.jar.

###### Forme d'un déploiement de site

Un site est déployé directement à l'URL indiquée dans le POM. Il n'y a donc pas d'arborescence "automatique" créée comme pour le déploiement release/snapshot. Afin d'avoir une arborescence correcte, si par exemple un serveur Web doit héberger le sites de plusieurs projets, il est possible de configurer l'URL avec des variables, par exemple le nom de l'artefact et la version :

<url>file:/home/maven/sites/${project.artifactId}/${project.version}</url>

##### Préconisations

Pour avoir des builds portables et configurables, il faut bien utiliser le système de configuration de Maven. On a déjà parlé de l'utilisation de variables dans l'URL de déploiement. La racine de l'URL peut aussi être une variable définie au niveau utilisateur, dans le fichier USER\_HOME/.m2/settings.xml. L'intérêt est d'avoir un build adapté selon l'utilisateur, tout en ayant un fichier POM identique :

* le serveur d'intégration continue définit le déploiement sur son système de fichier pour l'équipe de développement
* les développeurs définissent le déploiement sur leur machine ou sur le serveur d'intégration continue
* le chef de projet définit le déploiement sur le serveur à accès public et décide alors quand faire des release publiques

*Attention, ces réglages avancés ne doivent être faits que si cela est justifié*. En effet, dans le cadre de développement interne, le lieu du déploiement peut être fixe, sur un serveur d'intégration par exemple.

#### Faire une distribution (release)

Une fois le développement d'un logiciel (application, bibliothèque, framework...) achevé, il est temps de le distribuer. Maven permet d'automatiser cette tâche. Les tâches effectuées par Maven sont notamment :

* présence des éléments de configuration dans le POM (ex. : dépôt de sources)
* vérification de la consistance des dépendances (interdiction de faire une release s'il existe des dépendances vers des snapshot)
* modification du POM (ex. : mise à jour de la version)
* création d'un tag sur le système de versionning
* création de la distribution
* déploiement (sites et binaires)

Une distribution se passe en 2 temps : une phase de préparation et une phase de déploiement. La phase de préparation peut être simulée, afin de voir ce que se prépare à faire Maven sans pour autant effectuer les opérations (ex. : ajouter une version dans le système de versionning).

*Effectuer une release est un processus "manuel"*, dans le sens où cela est effectué à un moment donné, par un des développeurs. La création d'une release n'est pas faite dans le processus d'intégration continue (l'intégration continue peut faire en revanche des "release snapshot", c'est-à-dire des distributions faites à partir de la version en cours).

##### release:prepare

La phase de préparation prépare le terrain afin que *la distribution puisse être répétée à l'identique*. Cette phase est donc lancée une seule fois pour une release. Elle va effectuer les opérations qui permettent de figer le logiciel :

* création d'une version dans le dépôt de sources
* modification du POM pour par exemple indiquer explicitement les versions utilisées (dépendances et plugins)

Il est possible de simuler cette tâche en lançant la commmande suivante :

mvn release:prepare -DdryRun=true

Les informations permettant de figer le logiciel sont alors créées. Maven demande des informations pour effectuer correctement la release : numéro de la version à créer, TAG (CVS, SVN...), numéro de version suivante (pour la suite du développement). Des fichiers sont créés alors, notamment :

* pom.xml.tag : le POM de la release
* pom.xml.next : le POM de la version de développement suivante

On peut consulter ces fichiers pour s'assurer que les informations sont correctes. On peut effacer toute cette préparation avec la commande suivante :

mvn release:clean

La phase de préparation est réellement effectuée avec la commande suivante :

mvn release:prepare

##### release:perform

Cette tâche fait suite à release:prepare. Elle va effectuer un check out de la version (version créée par release:prepare), le build (compilation, tests unitaires, packaging, génération du site) et le déploiement (par défaut de la distribution et du site).

##### Ressources

[Site du plugin Maven Release](http://maven.apache.org/plugins/maven-release-plugin/)

Le déploiement étant une chose assez critique et ce document n'abordant que les principes, il est vivement conseillé de lire le chapitre 7.8 du livre "Better Builds With Maven" (inclus dans le projet CVS m2home).

#### Faire des distributions avec assembly

Le plugin release permet de "sortir" une version d'un logiciel. En plus d'effectuer les tâches fastidieuses concernant le versionning, il déploit un site et/ou les binaires du logiciel sur des dépôts. Le plugin assembly permet lui de faire un assemblage, c'est-à-dire une distribution prête à l'emploi, destinées aux utilisateurs. C'est ce que l'on peut télécharger sur de nombreux sites de logiciels libres : un ZIP contenant les binaires, les sources, les dépendances et la documentation.

assembly n'effectue aucune opération sur le serveur de sources, il se contente de faire des fichiers archives contenant l'arborescence décrite dans un fichier.

##### Utilisation de assembly

Il faut faire la configuration suivante dans le POM du projet :

<build>  
 <plugins>  
 <plugin>  
 <artifactId>maven-assembly-plugin</artifactId>  
 <version>2.0-beta-1</version>  
 <configuration>  
 <descriptor>mon-descriptor.xml</descriptor>  
 <finalName>nom-archive</finalName>  
 <outputDirectory>assembly/output</outputDirectory>  
 <workDirectory>target/assembly/work</workDirectory>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

Il faut ensuite écrire un fichier descripteur et lancer la commande suivante :

mvn assembly:assembly

##### Ecriture d'un fichier descripteur

Un fichier décrivant un assemblage a l'allure suivante :

<assembly>  
 <includeBaseDirectory>true</includeBaseDirectory>  
 <id>bin</id>  
 <formats>  
 <format>zip</format>  
 </formats>  
 <fileSets>  
 <fileSet>  
 <includes>  
 <include>README\*</include>  
 <include>LICENSE\*</include>  
 <include>NOTICE\*</include>  
 </includes>  
 </fileSet>  
 <fileSet>  
 <directory>target</directory>  
 <outputDirectory></outputDirectory>  
 <includes>  
 <include>\*.jar</include>  
 </includes>  
 </fileSet>   
 </fileSets>  
 <dependencySets>  
 <dependencySet>  
 <outputDirectory>lib</outputDirectory>  
 <unpack>false</unpack>  
 <scope>runtime</scope>  
 </dependencySet>  
 </dependencySets>  
</assembly>

On y décrit des fichiers que l'on veut mettre dans l'archive, avec des balises fileSets/fileSet. On peut aussi choisir d'inclure les dépendances du projet, avec la balise dependencySets/dependencySet. Dans les 2 cas, une balise outputDirectory permet de choisir le chemin dans l'archive.

On remarque aussi que l'on peut choisir un format avec formats/formats. Les formats disponibles sont : zip, gz, tar, tar.gz et tar.bz2. On peut mettre plusieurs formats, le plugin créera alors autant de fichiers.

La balise identifiant est ajoutée à la valeur de la balise finalName dans le POM pour former le nom du fichier. Le fichier généré a alors un nom de la forme : ${finalName}-${id}.${format}.

##### Avoir plusieurs fichiers de description

Il est possible d'écrire plusieurs fichiers de description selon ses besoins : distributions binaires, avec ou sans dépendances ; distributions des sources... On peut normalement préciser le fichier descripteur à utiliser avec la commande suivante :

mvn assembly:assembly -Ddescriptor=path/to/descriptor.xml

Malheureusement, cette option ne semble pas très bien fonctionner. On peut tout de même faire de la façon suivante pour rendre la configuration plus souple :

* dans le POM :

<configuration>  
 <descriptor>${assembly.descriptor}</descriptor>  
 <finalName>${project.artifactId}</finalName>  
 <outputDirectory>assembly/output</outputDirectory>  
 <workDirectory>target/assembly/work</workDirectory>  
</configuration>

On remarque que l'on utilise des variables pour le descripteur à utiliser (assembly.descriptor) et pour le nom de l'archive (le nom du projet).

* lors du lancement de la commande, on précise alors le descripteur à utiliser :

mvn assembly:assembly -Dassembly.descriptor=path/to/descriptor.xml

##### Cibles à lancer avant l'assemblage

assembly:assembly effectue un packaging avant l'assemblage. Si l'on veut par exemple aussi inclure dans l'assemblage le site Web du projet, il faut lancer cette cible (mvn site) avant assembly:assembly.

##### Ressources

[Site du plugin Maven Assembly](http://maven.apache.org/plugins/maven-assembly-plugin/)

[Mini-guide](http://maven.apache.org/guides/mini/guide-assemblies.html)

[Référence du fichier descripteur](http://maven.apache.org/plugins/maven-assembly-plugin/assembly.html)

### FAQ sur l'utilisation de Maven

#### Comment installer les sources et la javadoc?

Lors de l'exécution de la commande mvn install par défaut seul l'artéfact est installé. Il est possible d'y ajouter un jar des sources et un jar de la javadoc en positionnant la variable performRelease à la valeur true. [Documentation Maven associée](http://maven.apache.org/guides/introduction/introduction-to-the-pom.html)

mvn install -DperformRelease=true

#### Comment construire un projet Eclipse à partir des sources

Une commande maven permet de générer un projet Eclipse. Le projet créé a les dépendances définies dans le pom. Ces dépendances sont définie à l'aide d'une variable (classpath variable d'Eclipse) M2\_REPO. Cette variable doit pointer à la racine du repository maven à utiliser. De plus, pour toutes les dépendances ayant leur source ou leur javadoc présente dans le repository, ces éléments seront accessibles dans le projet. [Documentation Maven associée](http://maven.apache.org/plugins/maven-eclipse-plugin/)

Pour créer le projet  
mvn eclipse:eclipse

Pour créer le projet et tenter de télécharger les sources et/ou la javadoc.  
mvn eclipse:eclipse -DdownloadSources=true

#### Comment manipuler plusieurs projets d'un même workspace Eclipse?->

Il est possible de créer un « lien » entre les projets d’un même workspace Eclipse (ou d’un même répertoire). Dans le répertoire contenant les projets ajoutez un pom.xml déclarant autant de module qu’il y a de projet (l’ordre n’a pas d’importance).

<?xml version="1.0" encoding="UTF-8"?>  
 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>fr.mediapost</groupId>  
 <version>1.0-SNAPSHOT</version>  
 <artifactId>fwk</artifactId>  
 <packaging>pom</packaging>  
 <name>Framework Mediapost (pom parent)</name>  
 <url>http://www.mediapost.fr</url><modules> <module>mediapost-core</module> <module>mediapostFwkJEE</module> <module>mediapostFwkJEETest</module> <module>mediapostFwkJEEsecurite</module> <module>facelets-archetype</module> <module>MunicipostPos</module> <module>MunicipostPos\_facelets</module> </modules> </project>

Pour toutes commandes Maven exécutées sur ce pom, l’ensemble des projets seront concernées.

Lancer les tests de tous les projets  
mvn test

Installer les dernières versions dans le repository locale  
mvn install

Construire les projets Eclipse en faisant une dépendance au niveau projet (et non au niveau du jar)  
mvn eclipse:eclipse

### Mettre le numéro de version dans une application

On peut vouloir mettre le numéro de version dans une application, par exemple dans un fichier de ressources qui sera utilisé pour l'afficher sur un écran. Pour cela, on utilise 2 fonctionnalités de Maven : les variables pré-définies et le filtrage des ressources lors du build (principe identique à Ant). On peut en effet faire référence à des variables par exemple :

* pom.version
* pom.name
* pom.artifactId

On utilise la syntaxe ${ma.variable} dans un fichier pour signifier que l'on veut substituer la valeur de la variable lors du build. Prenons l'exemple d'un fichier d'internationalisation, qui se trouve dans un répertoire de ressources du projet, par exemple src/main/resources :

application\_name=${pom.name}  
application\_version=${pom.version}

Il faut activer le filtrage des ressources dans le POM pour que la substitution se fasse lors du build :

<build>  
 <resources>  
 <resource>  
 <directory>src/main/resources</directory>  
 <filtering>true</filtering>  
 </resource>  
 </resources>  
</build>

Lors du build, les valeurs seront remplacées, par exemple :

application\_name=Prototype Municipost  
application\_version=1.0-SNAPSHOT

Si ce fichier est un fichier i18n pour JSF, on peut afficher le nom et la version de l'application de la façon suivante :

<h:outputText styleClass="note" value="#{messages.application\_name} #{messages.application\_version}" />

## Vérifier la qualité du code

Un certain nombre d'outils permettent d'évaluer la qualité du code d'un projet. Intégrés à Maven, ils sont configurés, lancés automatiquement et leurs rapports inclus dans le site généré. Cela permet d'avoir toutes les informations sur le projet au même endroit.

Il existe de nombreux outils dans le monde Java, mais nous nous focaliserons sur les plus connus et sur ceux qui sont le mieux intégrés dans Maven 2. **Il est possible de se reporter au** pom.xml **du projet [mediapost-core](http://sunic.mediapost.fr/dsiwiki/index.php?n=Accueil.FrameworkMediapost#pomparent) (pom parent mediapost) pour retrouver les exemples d'intégration de ces outils dans Maven 2**. Tous les projets (typiquement les projets prototypes créés à partir des [Archetypes Mediapost](http://sunic.mediapost.fr/dsiwiki/index.php?n=Maven.ArchetypesMediapost)) dont le fichier pom.xml hérite de ce POM parent vont bénéficier de l'intégration de ces outils.

### Intégration des outils dans Maven 2

L'intégration dans Maven 2 d'un outil de qualité du code peut se faire à 2 endroits :

* dans les rapports : l'outil propose un ou des rapports lors de la génération du site Maven
* dans la phase verify : cette phase est importante car elle stoppe par exemple l'installation d'un projet dans le repository local (voir [le détail des diférentes phases du processus de construction Maven](http://sunic.mediapost.fr/dsiwiki/index.php?n=Maven.Lifecycle)). On peut donc ajouter les vérifications d'un outil d'analyse de code afin d'autoriser ou non l'installation du projet

#### Intégration des rapports

Cela se fait dans la balise reporting/plugins du pom.xml. Typiquement vous retrouverez cet exemple dans le pom.xml du projet [mediapost-core](http://sunic.mediapost.fr/dsiwiki/index.php?n=Accueil.FrameworkMediapost#pomparent) (pom parent mediapost):

<reporting>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-checkstyle-plugin</artifactId>  
 <configuration>  
 <configLocation>config/mediapost\_checks.xml</configLocation>  
 </configuration>  
 </plugin>  
 </plugins>  
</reporting>

On voit que l'on peut y ajouter des éléments de configuration propres au plugin entre les balises <configuration></configuration>. **Ces éléments de configuration (**<configuration>...</configuration>**) sont globaux : ils sont aussi valables pour la partie** verify **par exemple**. A noter que l'on peut surcharger des réglages utilisés pour le goal verify mais que cela n'est pas nécessaire si ces réglages sont identiques à ceux des rapports (reporting).

#### Intégration dans verify

Cela se fait dans la balise build/plugins :

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-pmd-plugin</artifactId>  
 <executions>  
 <execution>  
 <goals>  
 <goal>check</goal>  
 <goal>cpd-check</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

### Checkstyle

Checkstyle est un outil de développement permettant de vérifier l’adhérence d’une application Java à un standard d’écriture de code. Il automatise le processus de revue de code qui est une tâche importante mais particulièrement laborieuse. Cet outil est particulièrement adapté aux projets qui mettent en œuvre un cadre normatif.

Checkstyle est complètement configurable : il peut être paramétré pour répondre à n’importe quel standard d’écriture de code. Par défaut, il applique le standard d’écriture émis par Sun.

#### Principales Fonctionnalités de vérification

* Commentaires Javadoc
* Normes de nommage
* Entêtes de fichier
* Les imports
* Limitation de tailles
* Les espaces
* L’ordre des modificateurs
* Les blocs
* Les syntaxes
* La construction des classes
* Détection du code dupliqué
* Les métriques (complexités cyclométriques,…)
* Divers (indentation…)
* Options (préconisations J2EE)

#### Configuration

La configuration se fait par un fichier XML, dans lequel on définit les règles à appeler.

#### Configuration dans Maven 2

On rajoute Checkstyle au rapport Maven 2 de la façon suivante :

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-checkstyle-plugin</artifactId>  
</plugin>

Le fichier de règles utilisé sera celui de Sun. Si on veut préciser le sien, il faut le faire de la façon suivante :

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-checkstyle-plugin</artifactId>  
 <configuration>  
 <configLocation>mediapost\_checks.xml</configLocation>  
 </configuration>  
</plugin>

Le paramètre dans l'élément configLocation indique la localisation du fichier de règles. Il peut s'agir d'un chemin de fichier, d'une URL ou d'une ressource. S'il s'agit d'une ressource, contenue par exemple dans le fichier JAR d'un projet qui se trouve dans le dépôt Maven, il faut indiquer cette dépendance à Checkstyle et cela, dans la partie build/plugins du POM :

<build>  
 <plugins>  
...  
  
<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-checkstyle-plugin</artifactId>  
 <!-- dependances pour les fichiers de regles -->  
 <dependencies>  
 <dependency>  
 <groupId>fr.mediapost</groupId>  
 <artifactId>mediapost-quality</artifactId>  
 <version>1.0</version>  
 <scope>compile</scope>  
 </dependency>  
 </dependencies>  
</plugin>  
  
...  
  
 </plugins>  
</build>  
  
<reporting>  
 <plugins>  
  
...  
<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-checkstyle-plugin</artifactId>  
 <configuration>  
 <!-- une ressource se trouvant le projet mediapost-quality -->  
 <configLocation>config/mediapost\_checks.xml</configLocation>  
 </configuration>  
</plugin>  
...  
  
 </plugins>  
</reporting>

Typiquement vous retrouverez cet exemple de configuration dans le pom.xml du projet [mediapost-core](http://sunic.mediapost.fr/dsiwiki/index.php?n=Accueil.FrameworkMediapost#pomparent) (pom parent mediapost)

Remarquons dans cet exemple que :

* le fichier de configuration de checkstyle est défini au sein de la balise <configuration></configuration> contenue dans la balise <reporting></reporting>.
  + cette configuration de checkstyle sera donc également utilisée pour le goal maven verify
* le fichier indiqué config/mediapost\_checks.xml correspond à une ressource qui n'est ni dans le projet concerné ni dans le projet du pom parent, mais dans un projet annexe, le projet [mediapost-quality](http://sunic.mediapost.fr/dsiwiki/index.php?n=Accueil.FrameworkMediapost#pomqualite) qui est une dépendance du projet concerné. Rappelons que [mediapost-quality](http://sunic.mediapost.fr/dsiwiki/index.php?n=Accueil.FrameworkMediapost#pomqualite) est utilisé pour centraliser la définition des dépendances de base, des profils, des rapports des projets Mediapost.

#### Validation Checkstyle

Checkstyle peut être ajouté à la phase verify de Maven de la façon suivante :

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-checkstyle-plugin</artifactId>  
 <executions>  
 <execution>   
 <goals>  
 <goal>check</goal>  
 </goals>  
 </execution>  
 </executions>   
 </plugin>  
 </plugins>  
</build>

Il est possible d'ajouter un élément de configuration afin de ne pas avoir les même réglages que pour les rapports.

#### Ressources

[Checkstyle](http://checkstyle.sourceforge.net/)

[Plugin Maven 2 de Checkstyle](http://maven.apache.org/plugins/maven-checkstyle-plugin/)

### PMD et CPD

PMD est un outil comparable à Checkstyle, mais les règles qu’il contrôle sont des règles sémantiques et non syntaxiques. Il est donc complémentaire à Checkstyle, même si certains contrôles sont effectués par les deux outils. Par exemple, PMD intègre des règles de conception de plus haut niveau, comme la vérification des singletons, ou bien la vérification de l’utilisation des méthodes privées.

PMD dispose aussi d'un détecteur de duplication de code, CPD (Copy/Paste Detector), qui est inclus dans la distribution standard et les rapports.

#### Principales Fonctionnalités de vérification

* Blocs try/catch/finally/switch vides
* Variables locales, paramètres et méthodes privées inutilisées
* Déclarations if/while vides
* Expressions trop complexes
* Imports
* Strings
* Parenthèses
* Limitation de tailles
* Utilisation des logs
* Classes ayant des complexités cyclométriques importantes
* Détection du code dupliqué (module CPD)

PMD permet aussi de développer ses propres règles.

#### Configuration

La configuration des règles se fait à partir de fichiers XML appelés *ruleset*. PMD fournit un grand nombre de règles, regroupés par thèmes dans des rulesets. Quand on lance PMD, on choisit donc d'inclure certains rulesets.

#### Configuration dans Maven 2

Pour activer PMD dans Maven 2 :

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-pmd-plugin</artifactId>  
 <configuration>  
 <!-- les sources sont en 1.5 -->  
 <targetJdk>1.5</targetJdk>   
 <!-- pour CPD -->  
 <minimumTokens>100</minimumTokens>   
 </configuration>  
</plugin>

Une instruction dans la configuration indique que l'analyseur doit être lancé pour la version 1.5 de Java, afin de supporter les nouveautés syntaxiques (ex. : annotations, génériques).

Une autre instruction de configuration, minimumTokens, est réservée à CPD : elle permet de régler le nombre de signes (token) identiques et consécutifs à partir duquel CPD lance un avertissement de duplication. 100 signes correspondents approximativement à 5-10 lignes.

#### Choisir ses rulesets dans Maven

On peut choisir les rulesets (natifs à PMD) à inclure de la façon suivante :

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-pmd-plugin</artifactId>  
 <configuration>  
 <!-- les sources sont en 1.5 -->  
 <targetJdk>1.5</targetJdk>  
 <rulesets>  
 <ruleset>/rulesets/basic.xml</ruleset>  
 <ruleset>/rulesets/imports.xml</ruleset>  
 <ruleset>/rulesets/unusedcode.xml</ruleset>  
 <ruleset>/rulesets/finalizers.xml</ruleset>  
 <ruleset>/rulesets/migrating\_to\_15.xml</ruleset>  
 <ruleset>/rulesets/strings.xml</ruleset>  
 </rulesets>  
 </configuration>  
</plugin>

#### Validation PMD

PMD et CPD peuvent être ajoutés à la phase verify de Maven de la façon suivante :

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-pmd-plugin</artifactId>  
 <executions>  
 <execution>  
 <goals>  
 <goal>check</goal>  
 <goal>cpd-check</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

Il est possible d'ajouter un élément de configuration afin de ne pas avoir les mêmes réglages que pour les rapports.

#### Ressources

La liste exhaustive des règles (classées par ruleset) peut être trouvée sur le site Web de PMD.

[PMD](http://pmd.sourceforge.net/)

[Plugin Maven 2 de PMD](http://maven.apache.org/plugins/maven-pmd-plugin/index.html)

### JDepend

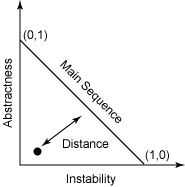
JDepend est un outil développé en Java qui analyse le bytecode d’un projet Java et calcule des métriques de qualité pour chaque package. JDepend est sensible à *l'extensibilité*, *la réutilisabilité* et *la maintenabilité* des sources.

Il est développé par Mike Clark de la « Clarkware Consulting, Inc ». Il est soumis à la licence « BSD License ».

L’outil JDEPEND permet de montrer les dépendances des packages et classes ainsi que les métriques de qualité associées à chacun d’eux.

#### Métriques

* Nombre de classes (CC & AC).
  + Le nombre de classes concrètes (CC) et de classes abstraites (et interfaces) (AC) est un indicateur d’extensibilité d’un package.
* Afferent coupling (Ca) où couplage par dépendance descendante.
  + Le nombre de packages qui utilisent les classes du package analysé peut être un indicateur de mauvaise gestion des dépendances ou être le signe que le package est le centre de l’application. Plus ce nombre est grand, plus il est nécessaire de fragmenter le package.
* Efferent coupling (Ce) où couplage par dépendance ascendante.
  + Le nombre de packages utilisés par les classes du package analysé. C’est un indicateur d’indépendance du code. Plus ce nombre est faible, mieux c’est.
* Abstractness (A) où degré d’abstraction.
  + Pourcentage de classes concrètes par rapport aux classes abstraites (et interfaces).
  + Proche de 0 : package concret
  + Proche de 1 : package abstrait.
  + Le degré d’abstraction d’un package doit tendre vers l’une ou l’autre des deux borne : 0 ou 1. Une valeur proche de 0.5 montrerait une mauvaise écriture du code.
* Instability (I) où degré de stabilité.
  + Indicateur de résilience du package : propriété de stabilité par rapport à la mise à jour d’autres packages.
  + Proche de 0 : package stable, proche de 1 : package instable.
  + Lié aux critères de dépendances ascendante et descendante.
  + I = Ce / (Ce+Ca)
* Distance à la Séquence Principale (Main Sequence : D)



* Cycles de dépendances : aucun cycle ne doit être levé par JDepend. Un cycle dans un projet met à mal sa modularité.

#### Intégration dans Maven 2

Le rapport de JDepend s'intègre au site Maven 2 de la façon suivante :

<plugin>  
 <groupId>org.codehaus.mojo</groupId>  
 <artifactId>jdepend-maven-plugin</artifactId>  
</plugin>

#### Ressources

[Site Web de JDepend](http://clarkware.com/software/JDepend.html)

[Plugin Maven 2 de JDepend](http://mojo.codehaus.org/jdepend-maven-plugin/)

### Cobertura

Cobertura est un outil qui permet de mesurer la couverture des tests unitaires. "Couverture" se traduit dans le cas de Cobertura par le pourcentage de code applicatif **accédé** par les tests unitaires. Cobertura calcule :

* le nombre de lignes exécutées
* le nombre de branches exécutées
* la complexité cyclomatique

Pour faire cela, Cobertura "instrumentalise" les sources : il les compile en y ajoutant des aspects qui seront appelés en même temps que les tests unitaires exécutent le code applicatif. Ces aspects enregistrent alors les appels et les journalisent.

*Attention : Le mode de fonctionnement de Cobertura fait que les tests unitaires sont exécutés deux fois lors d'un build Maven, ce qui peut augmenter fortement sa durée !*.

#### Intégration de Cobertura dans Maven 2

Cobertura s'intègre à Maven 2 de la façon suivante :

<plugin>  
 <groupId>org.codehaus.mojo</groupId>  
 <artifactId>cobertura-maven-plugin</artifactId>   
</plugin>

Il faut aussi impérativement ajouter ces paramètres dans build/plugins pour effacer le fichier d'instrumentalisation des sources à chaque build :

<plugin>  
 <groupId>org.codehaus.mojo</groupId>  
 <artifactId>cobertura-maven-plugin</artifactId>  
 <executions>  
 <execution>  
 <goals>  
 <goal>clean</goal>  
 </goals>  
 </execution>  
 </executions>   
</plugin>

#### Validation de Cobertura

Cobertura peut se greffer sur la phase verify de Maven de la façon suivante :

<plugin>  
 <groupId>org.codehaus.mojo</groupId>  
 <artifactId>cobertura-maven-plugin</artifactId>  
 <configuration>  
 <check branchrate="80" linerate="70" haltonerror="true"  
 totalbranchrate="80" totallinerate="70">  
 <regex pattern="com.example.reallyimportant.\*"   
 branchrate="80" linerate="90"/>  
 <regex pattern="com.example.boringcode.\*"   
 branchrate="40" linerate="30"/>  
 </check>  
 </configuration>  
 <executions>  
 <execution>  
 <goals>  
 <goal>clean</goal>  
 <goal>check</goal>  
 </goals>  
 </execution>  
 </executions>  
</plugin>

#### Ressources

[Cobertura](http://cobertura.sourceforge.net/)

[Plugin Maven 2 de Cobertura](http://mojo.codehaus.org/cobertura-maven-plugin/index.html)

### JavaNCSS

JavaNCSS est un outil développé en Java qui analyse le code source d’un projet Java et calcule des métriques de qualité. Il est sous licence « GNU General Public License ».

Il donne des métriques générales par projet, package, classe et méthode :

* nombre de package
* nombre de classes
* nombre de méthodes
* nombre d’instructions sans commentaire (Non Commenting Source Statements : NCSS)
* nombre de commentaires JavaDoc

L'intégration du rapport dans Maven 2 se fait de la manière suivante :

<plugin>  
 <groupId>org.codehaus.mojo</groupId>  
 <artifactId>javancss-maven-plugin</artifactId>  
 <version>2.0-beta-1</version>  
</plugin>

[Plugin Maven 2 de JavaNCSS](http://mojo.codehaus.org/javancss-maven-plugin/)

### QALab

QALab est un outil qui permet *d'aggréger les rapports d'autres outils dans le temps*. En effet, chaque outil propose des rapports très complets, mais à un instant donné. QALab permet, pour certains outils, de suivre l'évolution dans le temps de ces rapports, afin d'avoir une idée de l'évolution de la "santé" d'un projet.

Les outils supportés par QALab sont PMD, Checkstyle, Simian, FindBugs et Cobertura.

#### Limitation du support pour Maven 2

QALab repose sur plusieurs outils Open Source qui n'ont pas forcément tous la même réactivité. Cela fait que, actuellement, *on ne peut utiliser que PMD et Checkstyle avec le plugin Maven 2 de QALab*.

#### Principe de fonctionnement

QALab analyse les rapports (XML la plupart du temps) des outils qu'il supporte et stocke des données statistiques dans un fichier, qalab.xml (à la racine du projet). Ce fichier doit être créé et bien sûr conservé (dans le dépôt de sources).

#### Intégration dans Maven 2

L'intégration de QALab dans Maven 2 se fait de la manière suivante, d'abord dans build/plugins :

<plugin>  
 <groupId>net.objectlab</groupId>  
 <artifactId>maven-qalab-plugin</artifactId>  
 <version>2.1</version>  
 <executions>  
 <execution>  
 <id>checkstyle-merge</id>  
 <phase>site</phase>  
 <goals>  
 <goal>merge</goal>  
 </goals>  
 <configuration>  
 <handler>net.objectlab.qalab.parser.CheckstyleStatMerge</handler>  
 <inputFile>${project.build.directory}/checkstyle-result.xml</inputFile>  
 </configuration>  
 </execution>   
 <execution>  
 <id>pmd-merge</id>  
 <phase>site</phase>  
 <goals>  
 <goal>merge</goal>  
 </goals>  
 <configuration>  
 <handler>net.objectlab.qalab.parser.PMDStatMerge</handler>  
 <inputFile>${project.build.directory}/pmd.xml</inputFile>  
 </configuration>  
 </execution>  
 <execution>  
 <id>qalab-chart</id>  
 <phase>site</phase>  
 <goals>  
 <goal>chart</goal>  
 </goals>  
 <configuration>  
 <summaryOnly>false</summaryOnly>  
 </configuration>  
 </execution>  
 </executions>  
 <configuration>  
 <types>pmd,checkstyle</types>  
 </configuration>  
</plugin>

Puis dans reporting/plugins :

<plugin>  
 <groupId>net.objectlab</groupId>  
 <artifactId>maven-qalab-plugin</artifactId>  
 <version>2.1</version>  
 <reportSets>  
 <reportSet>  
 <reports>  
 <report>report</report>  
 </reports>  
 </reportSet>  
 </reportSets>  
</plugin>

#### Ressources

[QALab](http://qalab.sourceforge.net/)

[Plugin Maven 2 de QALab](http://qalab.sourceforge.net/maven2/index.html)

### Les outils à suivre

Les outils suivants n'ont aucun support ou alors ne sont pas encore satisfaisants pour Maven 2, néanmoins, il est prévisible que ce manque soit comblé dans les prochains mois :

* [Simian](http://www.redhillconsulting.com.au/products/simian/) : outil pour l'analyse de duplication. Son utilisation peut s'avérer redondante avec CPD mais il peut s'intégrer à QALab
* [FindBugs](http://findbugs.sourceforge.net/) : outil pour trouver des bugs dans le code Java, il complète bien PMD et Checkstyle
* [QALab](http://qalab.sourceforge.net/) : le support Maven 2 actuel n'est pas complet mais devrait s'améliorer

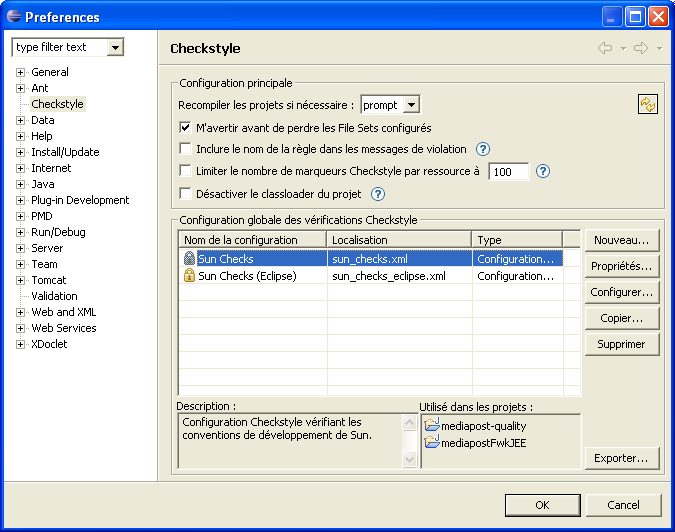
### Intégration dans Eclipse

Il est possible d'intégrer des outils directement dans Eclipse, cela afin que les erreurs soient corrigées directement, avant la génération du site Maven. L'intégration se limitera à Checkstyle et PMD.

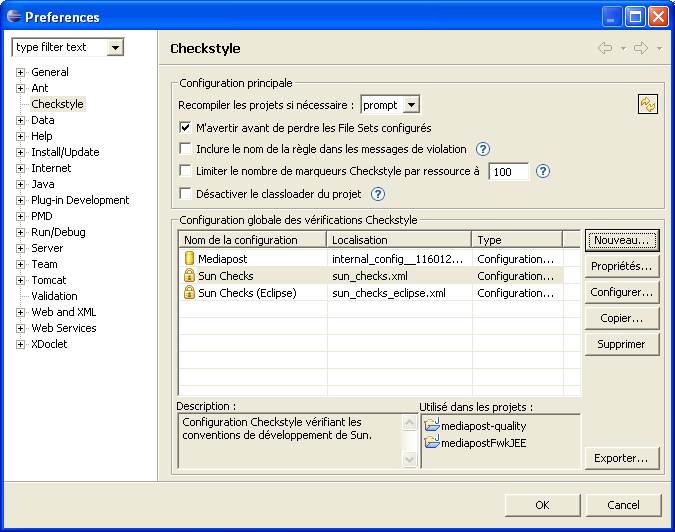
#### Checkstyle

Le plugin Checkstyle choisi est [Eclipse CS](http://eclipse-cs.sourceforge.net/). Il permet d'ajouter des fichiers de règles et d'activer la vérification sur un projet. Les erreurs (ou avertissements) apparaissent alors dans l'éditeur ou dans la vue "Problems".

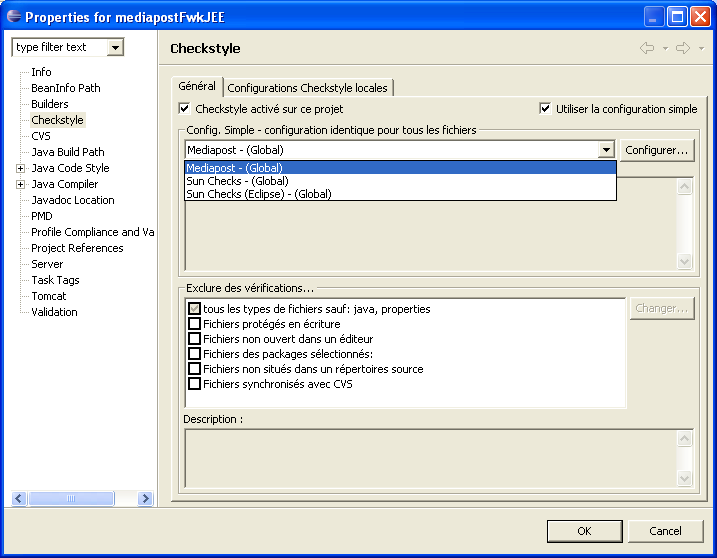
Le fichier de règles par défaut étant très strict, la première chose à faire est d'ajouter celui de Mediapost. Aller dans "Windows|Preferences|Checkstyle" :



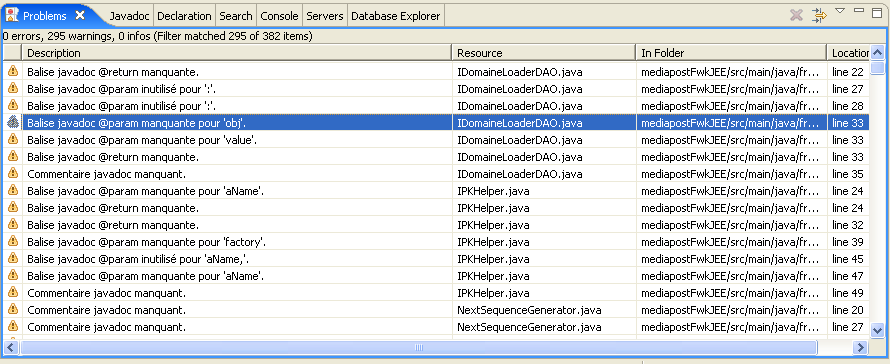
Cliquer sur "Nouveau", entrer un nom (par exemple "Mediapost") et faire "Importer" pour sélectionner le fichier de règles. Cliquer sur "OK". Le fichier de règles apparait alors dans la liste :



Il faut ensuite activer la vérification sur les projets, en cliquant bouton droit sur le projet, puis "Propriétés|Checkstyle". Il faut cocher "Checkstyle activé sur le projet" et choisir le fichier de règles :



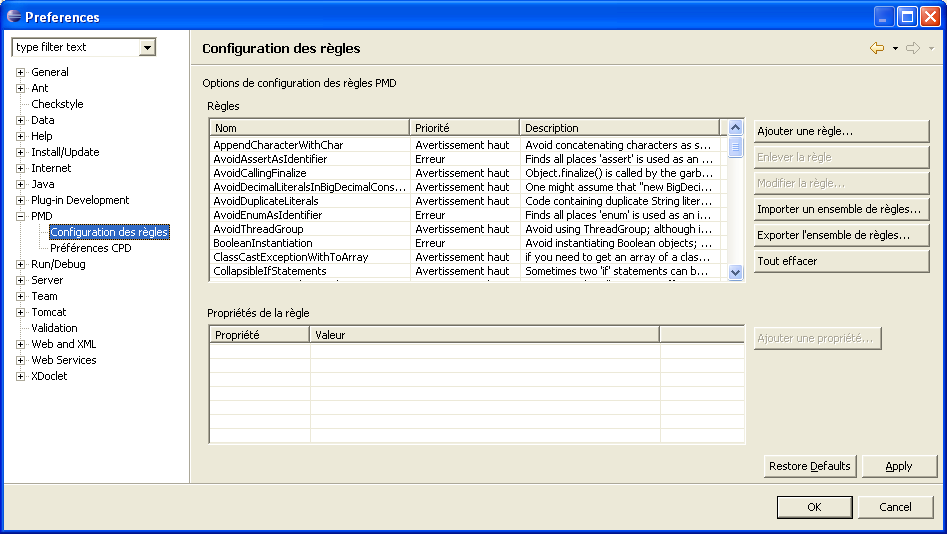
Le projet est alors recompilé et les violations de règles apparaissent dans la vue "Problems" :



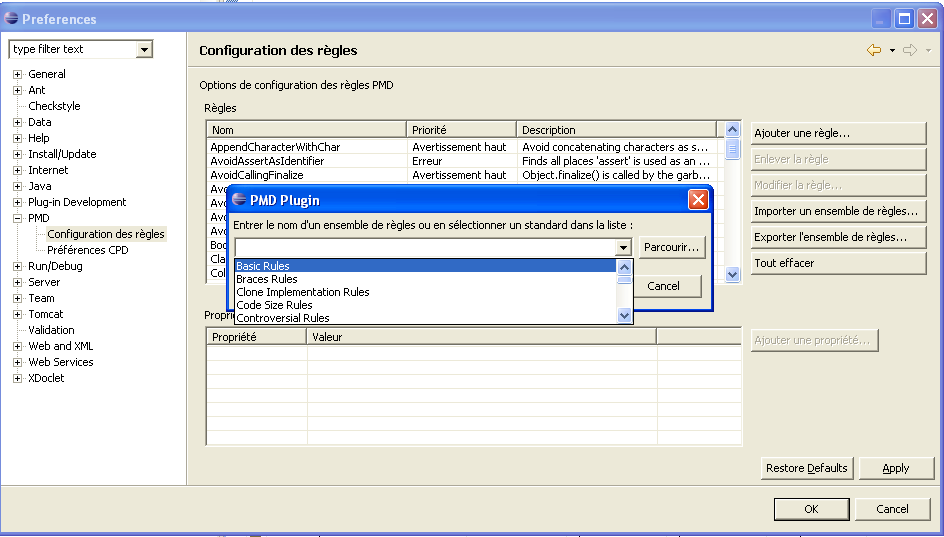
#### PMD

L'installation du plugin peut se faire en ligne en suivant [les instructions sur le site de PMD](http://pmd.sourceforge.net/integrations.html#eclipse).

Une fois le plugin installé il faut choisir les fichiers de règles dans "Windows|Preferences|PMD|Configuration des règles" :



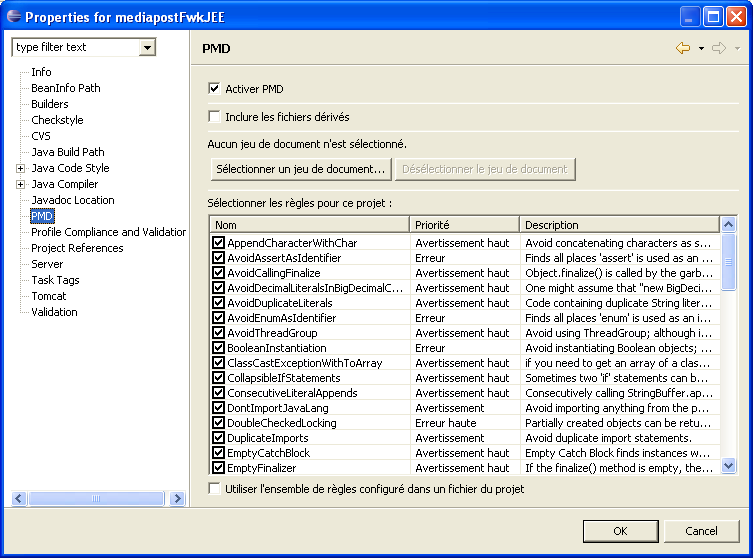
Il est conseillé d'effacer toutes les règles par défaut ("Tout effacer") et d'ajouter les règles ruleset par ruleset ("Importer un ensemble de règles") :



##### Utilisation du plugin

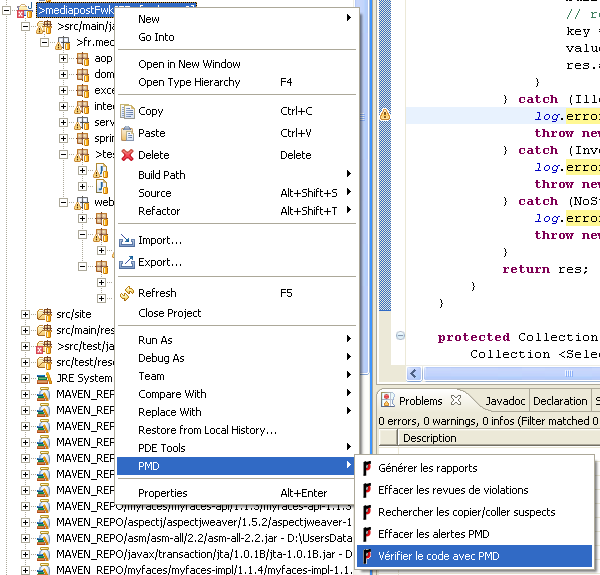
On peut utiliser le plugin de 2 manières :

* en activant PMD pour un projet (idem que pour Checkstyle), en cochant "Activer PMD" :

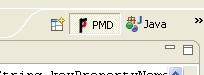


Les violations apparaissent alors dans la vue "Problems".

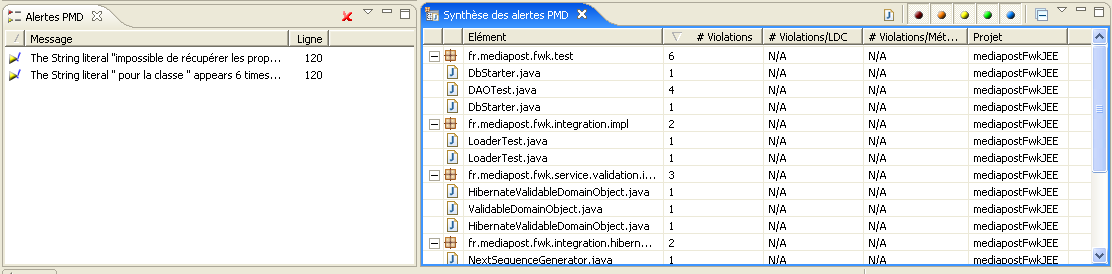
* en lançant explicitement PMD :



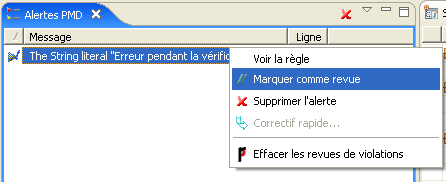
On passe alors dans la perspective PMD, ce qui est indiqué dans la barre des perspectives (en haut à droite) :



La perspective est en fait composée de 2 vues, qui permettent de voir les violations pour le fichier sélectionné (Alertes PMD) et pour l'ensemble du projet (Synthèse des alertes PMD) :



La vue "Alertes PMD" permet de désactiver la vérification d'une règle :



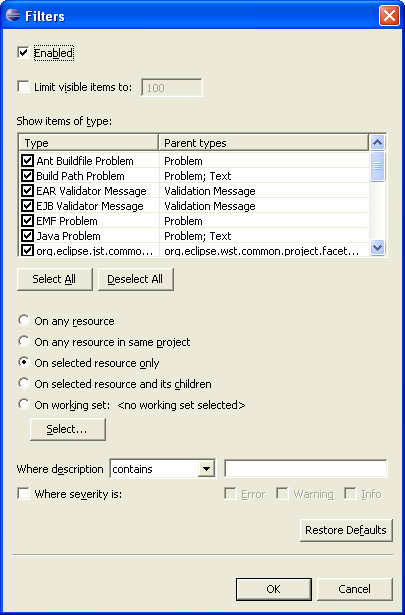
Ce qui provoque l'ajout d'un tag "NOPMD" sur la ligne correspondante :



Cela permet de supprimer la vérification PMD pour cette ligne, si l'erreur levée n'est pas pertinente, évitant ainsi de polluer les rapports. Le tag NOPMD peut bien sûr être ajoutée directement dans le code, à la main.

#### Conseils d'utilisations des plugins

Il est conseillé d'activer les plugins systématiquement, pendant l'édition du code. En revanche, il est intéressant alors de limiter les messages de la vue "Problems" à la ressource éditée (icône de réglages des filtres) :



Cela permet de se concentrer sur l'édition du fichier en cours et ne pas être pollué par les messages du projet ou pire, du workspace.

# Couche présentation

La couche présentation gère l'affichage de composants graphiques et l'interaction avec les utilisateurs. La spécification **JSF** (Java Server Faces) a été retenue pour la gestion de cette couche.

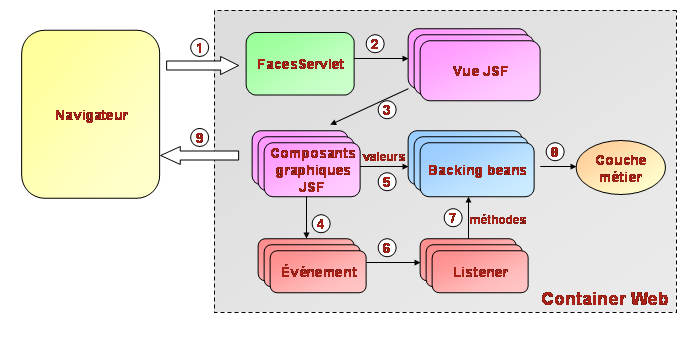
## Présentation générale

JSF propose un framework plus proche des applications clients-serveurs en introduisant :

* Des composants orientés UI (Interface Utilisateur)
* Des événements émis par ces composants UI et gérés par des objets listener

L'implémentation qui a été choisie est MyFaces, elle implémente à 100% la specification 1.1 de JSF  
et apporte des fonctionnalités supplémentaires par rapport à l'implémentation standard  
(intégration facilitée des Tiles par exemple) et notamment un certain nombre de composants graphiques.  
Tomahawk, Tobago et Trinidad (donation d'ADF Faces d'Oracle) sont des ensembles de composants étendus.  
MyFaces ne propose pas encore à ce jour (12/09/2006) d'implémentation de la spécification 1.2 de JSF,  
alors que celle de SUN RI est déjà sortie. Néanmoins la spécification 1.2 n'est sortie en version "Final Release" que le 11 mai 2006,  
la roadmap de MyFaces n'est pas fixée mais on peut espérer une implémentation MyFaces de JSF 1.2 d'ici la fin d'année 2006.

Voici le schéma de traitement d'une requête HTTP par le framework JSF.  
Il est nécessaire de bien comprendre chacune des étapes pour le développement d'applications web basées sur JSF.



## Quelles sont les meilleures pratiques en terme de design des backing beans ?

### Les beans managés par JSF

Le framework JSF permet de gérer lui-même les backing-beans, on parle alors de "beans managés par JSF" (JSF managed bean). La préconisation est justement de n'utiliser que des "beans managés par JSF" (sauf exceptions : voir ci-dessous). La spécification JSF parle également de "managed bean creation facility". Il devient alors possible de gérer de manière centralisée la déclaration et la création des beans, plutôt que de privilgéier un design où les beans sont créés à la volée dans les méthodes actions des backing-beans (ce qui peut entrainer des problèmes de performances et de maintenance)

Pour qu'un bean soit managé par JSF, vous devez le déclarer dans le fichier de configuration faces-config.xml. Exemple:

<faces-config>  
 <managed-bean>  
 <managed-bean-name>userBean</managed-bean-name>  
 <managed-bean-class>com.bankonet.jsf.beans.UserBean</managed-bean-class>  
 <managed-bean-scope>session</managed-bean-scope>  
  
 <!-- OPTIONNEL : les propriétés peuvent être managées -->  
 <managed-property>... </managed-property>  
 </managed-bean>  
 ...   
 </faces-config>

Il peut alors être stocké dans différents scopes (requête, session, application) et retrouvés par le framework JSF grâce cette la référence que vous précisez dans la balise <managed-bean-name>

Par exemple, les expressions JSF EL permettent de retrouver ce bean. Un tag inputText va permettre d'affecter (bind) la valeur de son composant à une propriété du backing-bean userBean. A noter que cette opération s'appelle le *[Value binding réel](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Binding)* et intervient notamment dans la phase *Update Model Values* (mise à jour des valeurs du modèle depuis les valeurs locales de chaque composant) du [cycle de vie du traitement des requêtes JSF](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.CycleVieRequeteJSF)

Ainsi l'expression suivante dans une page jsp...

<h:outputText value="#{userBean.name}"/>

... permet d'afficher le nom de l'utilsateur. JSF récupère automatiquement l'instance de l'objet de type com.bankonet.jsf.beans.UserBean stocké en session sous le nom userBean.

#### Initialiser des propriétés de beans managés par JSF... L'injection de dépendances pour les backing beans

Quand une page jsp référence un bean, le framework JSF l'initialise en fonction des éléments de configuration indiqués dans le fichier faces-config.xml. Il est en effet possible de personnaliser l'intialisation des propriétés des beans sans code additionnel (en utilisant les balises <managed-property/> )

Exemple:

<faces-config>  
 <managed-bean>  
 <managed-bean-name>userBean</managed-bean-name>  
 <managed-bean-class>com.bankonet.jsf.beans.UserBean</managed-bean-class>  
 <managed-bean-scope>session</managed-bean-scope>  
 <managed-property>  
 <property-name>company</property-name>  
 <property-class>java.lang.String</property-class>  
 <value>mediapost</value>  
 </managed-property>  
 <managed-property>  
 <property-name>department</property-name>  
 <property-class>java.lang.String</property-class>  
 <value>computing</value>  
 </managed-property>  
 </managed-bean>  
 ...   
 </faces-config>

Ainsi l'expression suivante dans une page jsp...

<h:outputText value="#{userBean.company}"/>

... permet d'afficher l'entreprise de l'utilsateur, qui est donc intialisée à la valeur mediapost.

Mais l'intérêt réel de l'initialisation des propriétés de beans managés par JSF n'est pas dans une telle "initialisation statique". Il peut être plus utile d'utiliser cette initialisation de propriétés lorsque vous désirez créer un bean qui référence d'autres beans, en d'autres termes un bean qui possède comme attribut(s) un (ou plusieurs) autre(s) bean(s) managé(s) par JSF. Prenons un exemple où on souhaite créer un bean représentant les informations d'un client, incluant son adresse, cette adresse étant un bean managé par JSF.

Exemple:

<faces-config>

<managed-bean>  
 <managed-bean-name>customerBean</managed-bean-name>  
 <managed-bean-class>com.bankonet.jsf.beans.CustomerBean</managed-bean-class>  
 <managed-bean-scope>request</managed-bean-scope>  
 <managed-property>  
 <property-name>adress</property-name>  
 <value>#{addressBean}</value>  
 </managed-property>  
 <managed-property>  
 <property-name>customerType</property-name>  
 <value>New</value>  
 </managed-property>  
 </managed-bean>  
 <managed-bean>  
 <managed-bean-name>addressBean</managed-bean-name>  
 <managed-bean-class>  
 com.mycompany.mybeans.AddressBean </managed-bean-class>  
 <managed-bean-scope>none</managed-bean-scope>  
 ...

</managed-bean>

...   
 </faces-config>

Cette déclaration permet de créer une instance du bean customerBean qui possède une propriété du bean addressBean représentant l'adresse géographique du client. On obtient ainsi un arbre de beans où le CustomerBean est la racine et le bean de type AddressBean est une feuille:

* La première déclaration customerBean crée un bean de type CustomerBean dans le scope request. Ce bean possède une propriété address. Cette propriété utilise l'élément value pour référencer un bean nommé addressBean
* La deuxième déclaration addressBean définit un bean mais ne le crée pas puisque son scope est à none. Ce scope permet de créé ce bean uniquement lorsqu'un autre objet le référence.

Lorsque un bean référence d'autres beans, attention à respecter certaines contraintes : vous ne pouvez pas référencer un bean qui a un cycle de vie plus court que le père puisqu'il spourrait être impossible de retrouver les ressources de ce scope lorsque l'objet est détruit. Par exemple, un objet de scope session ne peut pas référencé un objet de scope request. Les objets de scope none n'ont pas une durée de vie qui est gérée par le framework JSF, donc ils ne peuvent référencer que des objets de scope none. Voici un tableau vous permettant de respecter ces contraintes :

**Une objet de ce scope ==> peut référencer un objet de ce scope**  
none ==> none  
application ==> none, application  
session ==> none, application, session  
request ==> none, application, session, request

#### Dans quels cas les beans peuvent-ils ne pas être managés par JSF?

En théorie : jamais!

#### Comment récupérer un bean managé par JSF sans injection de dépendances?

Récupérer un bean managé en session par JSF à partir d'un autre bean (lorsque le premier n'est pas injecté au premier) se fait de deux manières: (labelBean est la valeur passée à la balise <managed-bean-name> dans le faces-config.xml)

* Première solution : En utilisant l’Expression Language JSF sans passer par l'ExternalContext: FacesContext fc = FacesContext.getCurrentInstance(); MyManagedBean bean = (MyManagedBean ) fc.getApplication().createValueBinding("#{labelBean}").getValue(fc);
* Deuxième solution : Ou en attaquant la session directement et en passant par l'ExternalContext: FacesContext fc = FacesContext.getCurrentInstance(); MyManagedBean bean = (MyManagedBean ) fc.getExternalContext().getSessionMap().get('labelBean');

Voir [la gestion du contexte applicatif](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Contexte)

Il peut-être préférable d'utiliser la deuxième solution, puisque moins fastidieux à écrire, par contre si ces lignes de code sont encapsulées dans un Helper dédié, le développeur ne sera pas confronter à la complexité du code, il peut être alors préférable d'utiliser la première solution puisque sa grande singularité (à la différence de la deuxième) réside dans le fait que **le bean est créé s'il n'existe pas**! Un Helper devrait donc être développé dans le framework Mediapost.

### Les scopes des beans

Vous pouvez préciser différents scopes pour chaque backing-bean managé par JSF grâce à la balise <managed-bean-scope/>. Quatre valeurs sont possibles pour ce scope :

* none : le bean est alors instancié chaque fois qu'il est référencé, il ne saura sauvegardé dans aucun scope.

Lorsque un bean managé par JSF référence un autre bean managé par JSF,le second bean référencé peut avoir un scope à none si il est supposé être créé uniquement lorsqu'il est référencé (voir ci-dessus)

* request
* session
* application

Nous préconisons de disposer d'un backing-bean de scope **session** pour les traitements ET les données sous-jacents à un écran: voir exemple détaillé: [Comment implémenter un use-case standard d'un écran de recherche aboutissant à une liste de résultat puis à une fiche détaillée?](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.UsecaseListDetailHowto) L'exception à cette régle réside dans le cas où il est nécessaire d'embarquer des UIComponent dans les backing beans : dans ce cas le backing devra être de scope **request** puisque les UIComponent ne sont pas serializables. Ce cas s'appelle le [component-binding](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Binding)

### L'initialisation des beans

Il n'est pas préconisé d'utiliser les constructeurs pour initialiser les beans JSF.

## Comment implémenter un simple use case ?

### 1- Déclaration du fichier web.xml

Il est nécessaire de spécifier les parametres suivants pour la configuration de JSF (fichier WEB-INF/web.xml): voir [Configuration JSF](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.ConfigurationJSF)

### 2- Les règles de navigation entre les pages

Il est nécessaire d'écrire des règles de navigation (fichier WEB-INF/classes/faces-config.xml).

On suppose que toute vue ou backing bean appelant le from-outcome **rechercheClient** redirigera l'utilisateur vers la vue **/jsps/rech\_client.jsp**

<navigation-rule>  
 <from-view-id>\*</from-view-id>  
 <navigation-case>  
 <from-outcome>rechercheClient</from-outcome>  
 <to-view-id>/jsps/rech\_client.jsp</to-view-id>  
 </navigation-case>  
 </navigation-rule>

### 3- Création des Javabeans

Le Backing bean doit être développé comme un simple javabean:

public class ClientBean extends MunicipostAbstractBean{  
  
 private String nom;  
 private String raisonsociale;  
 private String siret;  
 private String coclico;  
 private List clientListe;  
 private RcClient detailClient;  
  
 public List getClientListe() {  
 return clientListe;  
 }  
  
 public void setClientListe(List clientListe) {  
 this.clientListe = clientListe;  
 }  
  
 // autres getters/setters...  
  
 }

### 4- Déclaration des Javabeans (fichier WEB-INF/classes/faces-config.xml)

<faces-config>  
 <managed-bean>  
 <managed-bean-name>clientBean</managed-bean-name>  
 <managed-bean-class>fr.mediapost.municipost.web.backingbean.ClientBean</managed-bean-class>  
 <managed-bean-scope>session</managed-bean-scope>  
 </managed-bean>   
 <navigation-rule> … </navigation-rule>   
 </faces-config>

### 5- Création des JSP

Il est nécessaire d'utiliser les taglib JSF (préfixes h pour la taglib html et f pour la taglib core).

Il est nécessaire pour gérer correctement l'internationalisation de ne pas utiliser de libellé en dur dans les JSP mais d'utiliser un fichier de ressources (*resource bundle*).

On suppose qu'un fichier messages.properties existe à la racine du répertoire source de l'application (par exemple pour la prototype : MunicipostPos/src/main/java/messages.properties).

Il est nécessaire de le déclarer dans la JSP (tag <f:loadBundle />) et dans le fichier WEB-INF/classes/faces-config.xml.

Notons que la déclaration du tag <f:loadBundle /> est faite une seule fois dans la page *layout* des tiles : voir [Les tiles](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Tiles)

Voici un exemple de déclaration de ce fichier *resource bundle* (fichier WEB-INF/classes/faces-config.xml):

<application>  
 ...  
 <message-bundle>messages</message-bundle>  
 <locale-config>  
 <default-locale>fr</default-locale>  
 <supported-locale>en</supported-locale>  
 </locale-config>  
 </application>

Au sein de toute jsp, il est ainsi possible d'utiliser l'Expression Language pour récupérer un libellé:

<h:outputText value="#{messages.label\_welcome}"/>

Attention, il n'est pas possible d'utiliser un "." pour la clé du libellé, préférez les "\_" en conservant la même notation. En effet, si vous utilisez un "." il faudra faire appel à la notation suivante :

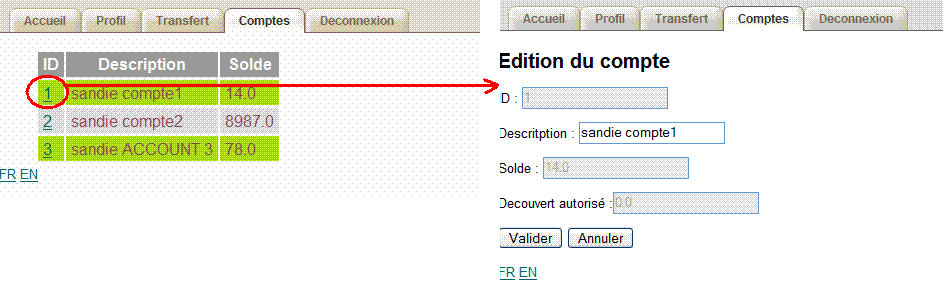
<h:outputText value="#{messages.['label.welcome']}"/>

Exemple de la page d'accueil:

<%@ taglib uri="<http://java.sun.com/jsf/html>" prefix="h"%>  
 <%@ taglib uri="<http://java.sun.com/jsf/core>" prefix="f"%>  
 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  
 <html>  
 <head>  
 <meta http-equiv="Refresh" content="1800;url=timeOutException.htm">   
 <meta content="text/html; charset=ISO-8859-15" http-equiv="content-type">  
 <link rel=stylesheet href="style/stylemunicipost.css" type="text/css">  
 <title>Application Municipost</title>  
 </head>  
 <body class="body">  
 <f:view>  
 <f:loadBundle basename="messages" var="messages" />  
 <h:form>  
 <h:outputText value="#{messages.label\_welcome}"/>  
 <h:commandButton action="rechercheClient" value="#{messages.label\_action\_prototype}" />  
 </h:form>  
 </f:view>

## Comment implémenter un use-case standard d'un écran de recherche aboutissant à une liste de résultat puis à une fiche détaillée ?

### 1- Premier use case : le détail du compte n’affiche pas les crédits du compte



**liste des comptes au détail d'un compte**

Un seul backing bean, AccountBeanList, permet de gérer les traitements et les données des pages. Ce packing bean:

* possède un attribut compteEdit de type Compte
* est de portée session puisqu'il va permettra le gérer le compte de l'écran de la liste jusqu'à la validation de la modification du détail.

Extrait de accountList.jsp (affichage liste des comptes):

<h:form>  
 …  
 <h:dataTable id="accountList"  
 headerClass="table\_th"  
 rowClasses="table\_tr1,table\_tr2" value="#{accountListBean.listAccounts}"   
 var="item">   
 <h:column>  
 <f:facet name="header">  
 <h:outputText value="#{labels.accountId}"/>   
 </f:facet>  
 <h:commandLink action="#{accountListBean.selectAccount}">  
 <h:outputText value="#{item.id}"/>  
 </h:commandLink>  
 </h:column>  
 <h:column>  
 <f:facet name="header">  
 <h:outputText value="#{labels.accountDescription}"/>  
 </f:facet>  
 <h:outputText value="#{item.description}"/>  
 </h:column>  
 <h:column>  
 <f:facet name="header">  
 <h:outputText value="#{labels.accountSolde}"/>  
 </f:facet>  
 <h:outputText value="#{item.solde}"/>  
 </h:column>  
 </h:dataTable>   
 …  
 </h:form>

Pour afficher le détail du compte :

* La récupération de l’instance du compte en request est automatique avec le composant UICommand ( <h:commandLink/> ou <h:commandButton/>. Il suffit d'utiliser une variable pour itérer sur les éléments de la datatable, cette variable est nommée item dans l'exemple de code ci-dessus (var="item"). Ainsi, dès que l'utilisateur clique sur un compte, une instance de ce compte est placée en scope request sous le nom de variable item
* Une seule ligne de code dans la méthode action AccountBeanList.selectAccount() permet de récupérer en request l'instance du compte qui a été choisi dans la liste des comptes (sous le nom de variable item)
* Il n'est pas nécessaire d’interroger à nouveau la base pour récupérer une autre instance du compte, puisqu’on ne veut pas afficher les dépendances du compte (par exemple nous ne voulons pas dans ce use-case afficher les crédits du compte)
* Sur l’écran d’affichage du détail d’un compte : le bouton de type reset permet d’annuler la modification
* Lorsque la validation du formulaire du détail est éronnée, l'instance du compte n'est pas modifiée. En effet dans le cycle de vie de traitement d'une requête JSF, la phase de validation est AVANT la phase "Update Model values" permettant de mettre à jour le backing-bean à partir des valeurs des composants graphiques de la page. Imaginons par exemple qu'un tag <f:validateLength /> est positionné à l'intéreur d'une balise <h:imputText /> de l'une des données du formulaire. Si la taille de l'input text n'est pas respectée par l'utilisateur, la validation va lever une erreur et engendrer un rendu immédiat de la page sans mettre à jour le backing bean.

Extrait de AccountBeanList.java (affichage du détail du compte):

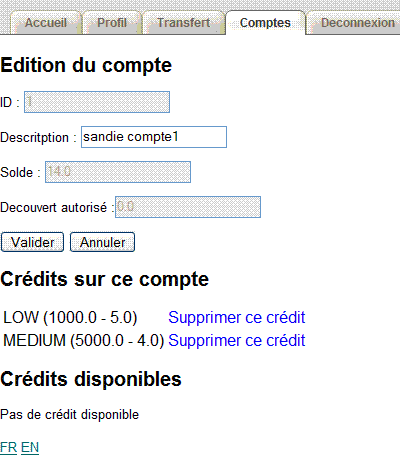
private Compte compteEdit;  
 …  
 public String selectAccount() throws FonctionnelleException {  
 String result = null;  
 // get facescontext and externalcontextFacesContext fc = FacesContext.getCurrentInstance();ExternalContext ec = fc.getExternalContext();   
 // get selected account from request  
 Compte compte = ((Compte)ec.getRequestMap().get("item"));  
  
 this.setCompteEdit(compte);  
 // update result  
 result = "accountDetail";   
 return result;  
 }

Extrait de accountDetail.jsp (affichage du détail du compte):

<h:form>  
 …  
 <h:outputText value="#{labels.editAccountId}"/> <h:inputText id="accountId" value="#{accountListBean.compteEdit.id}" disabled="true"/>  
 <h:outputText value="#{labels.editAccountDescritption}"/> <h:inputText id="accountDescription"  
 value="#{accountListBean.compteEdit.description}" required="true"/> <h:message for="accountId" />  
 …  
 <h:outputText value="#{labels.editAccountBalance}"/> <h:inputText id="accountBalance" value="#{accountListBean.compteEdit.solde}" disabled="true"/>  
 …  
 <h:commandButton id="submitbutton" value="#{labels.buttonSubmit}" action="#{accountListBean.updateAccount}" type="submit" />  
 <h:commandButton id="resetbutton" value="#{labels.buttonReset}" type="reset" />  
 …  
 </h:form>

### 2- Deuxième use case : le détail du compte affiche les crédits du compte

Imaginons maintenant qu'il soit nécessaire d'afficher certaines dépendances du compte, par exemple ses crédits:



**Détail d'un compte avec ses crédits**

Il est donc nécessaire d'interroger la base de données dans la méthode action AccountBeanList.selectAccount()

Extrait de AccountBeanList.java (affichage du détail du compte):

private Compte compteEdit;  
 …  
 public String selectAccount() throws FonctionnelleException {  
 String result = null;  
 // get facescontext and externalcontextFacesContext fc = FacesContext.getCurrentInstance();ExternalContext ec = fc.getExternalContext();   
 // get selected account  
 Integer id = ((Compte)ec.getRequestMap().get("item")).getId();  
 Compte compte = this.getServiceFactory().getBankService().selectCompte(id);   
 this.setCompteEdit(compte);  
 this.setListCreditsDuCompte(new ArrayList<Credit>(compte.getCredits()));  
 // get credits disponibles  
 Collection creditsDisponibles = this.getServiceFactory().getBankService().getCreditsDisponibles(compte);  
 this.setListCreditsDisponibles(creditsDisponibles);  
 // update result  
 result = "accountDetail";  
 return result;  
 }

## Comment gérer la validation ?

La validation concerne les données saisies par l'utilisateur.

JSF propose des composants permettant de gérer cette validation. Les composants concernés sont les composants UI qui implémentent l’interface EditableValueHolder, à savoir par exemple UIInput et ses dérivés

Par défaut, la validation ne se fait que localement au composant. Pas d'accès direct à la valeur des autres composants.

### Validation simple :

* Par l'attribut required du composant UIInput.

La valeur saisie pour ce composant est requise

* Par des tags de validation standards.

Une implémentation de JSF offre des validateurs standards

#### L'attribut required :

* L'implémentation de JSF vérifie que la valeur est non-nulle et de longueur supérieure à 0
* Si true, cette validation est la première effectuée

Exemple:

<h:inputText value="#{clientBean.login}" required="true" />

#### Tags de validation standards

Une implémentation de JSF doit fournir ces tags standards. Exemple:

<f:validateLength mininum='5' maximum='8'/>

Ces tags peuvent être utilisés par encapsulation dans le composant à valider :

<h:inputText value="#{clientBean.raisonsociale}" id="raisonsociale" required="false">  
 <f:validateLength maximum="8" />  
 </h:inputText>

Myfaces/Tomahawk propose d'autres types de tag pour les validations syntaxiques:

* validateCreditCard
* validateUrl
* validateEmail
* validateEqual
* validateRegExpr

Pour utiliser ces tags spécifiques de validation, il ne faut pas oublier de déclarer la taglib.

Exemple de l'utilisation du tag de validation d'adresse email:

<%@ taglib uri="<http://myfaces.apache.org/tomahawk>" prefix="t" %>  
 ...  
 <h:inputText id="email2"  
 value="#{clientBean.email}"  
 required="true">  
 <t:validateEmail />  
 </h:inputText>

### Validation avancée :

* Par l'attribut *validator* du composant UIInput

L'attribut *validator* précise la méthode qui validera la donnée

* *Par la méthode validate() du composant UIInput (transparent pour le développeur, cette méthode est automatiquement appelée à un moment spécifique dans le traitement de la requête JSF)*
* Par des tags de validation créés par le développeur

#### Appel d'une méthode dédiée

L'attribut *validator* permet de spécifier la méthode du backing bean qui sera appelée pour réaliser la validation.Expression Language (EL) est utilisé pour réaliser cet appel au sein de la JSP.

<h:inputText value="#{clientBean.nom}" validator="#{clientBean.validate}"/>

La signature de la méthode est fixe, à l'exception de son nom qui peut varierEn cas d'erreur de validation, la méthode doit lancer une ValidatorExceptionLe cas échéant, le champ est considéré comme valide.

Exemple de méthode dédiée à la validation:

public class ClientBean {  
  
 public void validate(FacesContext context, UIComponent component, Object value) throws ValidatorException {  
 String nom = (String) value;  
 if (…)  
 {  
 throw new ValidatorException(new FacesMessage("login non valide"));  
 }  
 }   
 }

#### Extension de JSF

Il est nécessaire d'étendre légèrement JSF puisque:

* Par défaut, les champs de formulaires sont validés séquentiellement.
* Par défaut, pas de validation entre plusieurs champs dépendants (cette fonctionnalité est annoncée en JSF 1.2)

Ex : Une date de début qui doit être antérieure à une date de fin.

La solution pour résoudre ce problème consiste à déclarer un champs caché dédié à la validationIl doit être déclaré après tous les champs "validables"

Exemple:

<h:outputText value="#{messages.datedebut}" />  
 <h:inputText id="dateDebut" value="#{dateBean.dateDebut}" />  
 <h:outputText value="#{messages.datefin}"/>  
 <h:inputText id="dateFin" value="#{dateBean.dateFin}" required="true" />**<h:inputHidden id="validate" value="needed" required="true" validator="#{dateBean.validateDates}"/>** <h:commandButton action="#{dateBean.process}" value="Valider" />

La copie des données du composant au backing bean n'a pas encore eu lieu (voir cycle de vie du traitement d'une requete):

* Il faut donc récupérer la valeur portée par les composants graphiques
* Si le formulaire n'est pas valide, la copie des données du composant au backing bean n'aura pas lieu

Exemple:

public class DateBean {  
  
 public void validateDates(FacesContext context, UIComponent component, Object   
 value) throws ValidatorException {UIInput dateDebutInput = (UIInput) component.findComponent("dateDebut");UIInput dateFinInput = (UIInput) component.findComponent("dateFin");  
 Date dateDebut = (Date) dateDebutInput.getValue();  
 Date dateFin = (Date) dateFinInput.getValue();  
 if (dateDebut.after(dateFin)) {  
 String labelError = … // Construction du label  
 throw new ValidatorException(new FacesMessage(labelError));  
 }  
 }   
 }

### Messages d'erreurs :

#### Déclaration

Les fichiers contenant les messages d'erreur par défaut sont placés dans l'implémentation JSF :

* javax/faces/Messages.properties
* javax/faces/Messages\_en.properties

Il est bien évidemment conseillé de redéfinir ses propres fichiers de ressources:

* 1- Créer son fichier de messages (par exemple à la racine du répertoire source (voir prototype fichier :  
  /src/main/java/messages.properties)
* 2- Le déclarer dans le fichier de config :

<application>  
 <message-bundle>messages</message-bundle>  
 ...  
 </application>

* 3- surcharger les messages d'erreurs du fichier par défaut:  
  Exemple d'extrait du fichier de messages redéfini:

# Component Errors  
 javax.faces.component.UIInput.REQUIRED=Erreur de validation: Valeur requise.  
 javax.faces.component.UISelectOne.INVALID=Erreur de validation: Valeur non valide.

#### Affichage du message d'erreur

* Solution 1 : Afficher les messages de tous les composants

<h:messages/>

* Solution 2 : Afficher le message d'erreur d'un composant :

<h:message for=*idComposant*/>

## Comment gérer la conversion ?

Dans les applications Web, la conversion entre les paramètres HTTP et l'objet domaine (bean transverse aux couches)  
est une fonctionnalité cruciale.  
Les frameworks web ne fournissent classiquement que des convertisseurs primitifs (ex: Struts).  
Mais pour la majorité des applications web, il est nécessaire de disposer d'une heuristique générique pour gérer la conversion des objets domaine. En effet, un anti-pattern serait que chaque objet domaine dispose d'un convertisseur spécifique : l'effort de développement et de maintenance serait très fort.

Avec JSF, il est possible d'utiliser la notion de convertisseur pour justement disposer d'une solution générique  
pour ce type de conversion. Nous allons d'abord apporter quelques précisions sur les convertisseurs de JSF  
pour ensuite présenter notre solution concernant la conversion des objets domaine.

### Les convertisseurs standards JSF

Un convertisseur transforme un attribut de type String d'une requête en un objet Java.  
Une implémentation de JSF doit fournir ces convertisseurs standards.  
Par exemple : DateTimeConverter convertit un String en un DateTime.

Il existe plusieurs façons d'utiliser les convertisseurs :

* 1. Conversion automatique: le composant UI « devine » le convertisseur à utiliser,  
  d’après le type de la valeur de l'attribut du bean  
  Exemple de JSP:

<h:inputText value="#{clientBean.age}"/>

Exemple de BackingBean:  
public class ClientBean{ private int age; }

* 2. Utilisation de l’attribut 'converter' : l’id logique du convertisseur à utiliser est  
  précisé manuellement dans un attribut du composant UI  
  Exemple de JSP:

<h:inputText value="#{clientBean.age}"   
 converter="javax.faces.Integer" />

* 3. Utilisation du tag <f:converter> : l’id logique du convertisseur à utiliser est  
  précisé manuellement dans un tag imbriqué dans le composant UI  
  Exemple de JSP:

<h:inputText value="#{clientBean.age}">  
 <f:converter converterId="javax.faces.Integer"/>  
 </h:inputText>

**Préconisation : il est plus souhaitable pour des conversions simples, comme la transformation   
en types primitifs, de laisser le composant UI récupérer automatiquement le convertisseur à utiliser.**

Pour les conversions plus complexes, comme la conversion en Double ou Date, il est conseillé d'utiliser  
les convertisseurs standards comme DateTimeConverter et NumberConverter  
Des attributs permettent de les configurer  
Exemple de JSP:

<h:outputText value="#{clientBean.dateNaissance}">  
 <f:convertDateTime dateStyle="full" locale="fr"/>  
 </h:outputText>

Exemple de BackingBean:

public class ClientBean{  
 private java.util.Date dateNaissance;  
...  
 }

### Exceptions

Si la conversion échoue, une ConverterException est levée  
Pour un composant UI de type UIInput, le message ayant pour clef "javax.faces.component.UIInput.CONVERSION" est affiché. Il est préconisé de surcharger ce message d'erreur dans le fichier de message propre à l'application. Voir [Gestion des messages d'erreur](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Validation#messageErreur)

## Comment utiliser les principaux tags/composants ?

### Modèle de Composant Graphique

Une des particularité de JSF est l'utilisation de composants graphiques, similaires à ceux du framework .NET

Le modèle de ces composants est agrégé en un **arbre des composants**

Chaque composant rend compte des interactions dont il fait l’objet via des événements spécifiques et des écouteurs (=« Listeners ») spécifiques : voir [Les Evénements](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Evenement)  
Il est possible d'étendre ce modèle de composants en développant de manière spécifique de nouveaux composants

Chaque composant est **stateful** : l’état des composants est conservé pendant une conversation entre l’utilisateur et le système.

Chaque composant est indépendant de son rendu : on a donc une séparation comportement/présentation ce qui assure une certaine versatilité grâce à cette indépendance à un langage de présentation spécifique (HTML, WML, etc.)

Chaque composant est instancié de façon transparente depuis la page. Celle-ci sert de patron à la construction des vues  
Elle utilise des librairies de tag spécifiques. A chaque Tag correspond donc un composant graphique.  
Par exemple le tag <h:commandButton/> correspond au composant UICommand  
Extrait d'une entête de page:

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>  
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>  
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t" %>  
<%@ taglib prefix="tiles" uri="/WEB-INF/classes/tlds/struts-tiles.tld"%>

Le traitement d’une page produit un arbre de composants graphiques (core, html, etc.)  
Chaque composant est manipulé à travers les interactions du client  
Il est source des événements à destination des listeners  
Il délègue:

* la conversion de données : à des converters
* la validation : à des validators
* la présentation : à des renderers

On appelle le lien des composants graphiquesavec les composants métier : value-binding

#### Modèle de Rendu

Le framework JSF assure un découplage entre les fonctionnalités du composant et son rendu pour un client particulier HTML, WML, …

1 composant = 1..n renderers

Ex : la sélection d’un élément parmi une liste en HTML peut se faire à travers plusieurs modèles de rendu :

* Drop down
* Radio buttons
* List box

Le RenderKit correspond à une association entre

* les classes de composant et
* les tags spécifiques à un type de client particulier

### L'écriture des pages

#### Support EL

EL: l’Expression Language  
Il s'agit d'un langage simple et flexible d’accès aux données  
La notation du EL standard est : ${…}  
Elle est utilisée pour encadrer les expressions dynamiques.  
Tout objet présent dans le scope de la JSP peut être utilisé comme variable dans une expression EL.

Attention : **La notation du EL JSF est #{...}, et non ${...}**  
En effet, JSF 1.1 ne gère pas le scope page. Il s'agit donc du fonctionnement de l'implémentation 1.1 de JSF  
A noter également que les EL de JSF ne sont pas interprétés par le moteur JSP alors que c'est le cas des EL standards.  
Les implémentations 1.2 utilisent l'EL unifié avec au choix l'utilisation de #{} ou ${} dans les composants JSF.

Attention avec EL:

* Il n'y a pas de typage et les conversions automatiques (nombres <-> String)
* L'évaluation des expressions est effectuée à l'exécution
* Il n'y a pas de vérification syntaxique, attention aux fautes de frappe : il y aura une erreur à l'exécution!

##### Accès à une variable

#{client} :

* recherche l'attribut client dans les contextes request, session, application
* retourne l'attribut dès qu'il est trouvé, sinon retourne null

##### Accès à une propriété

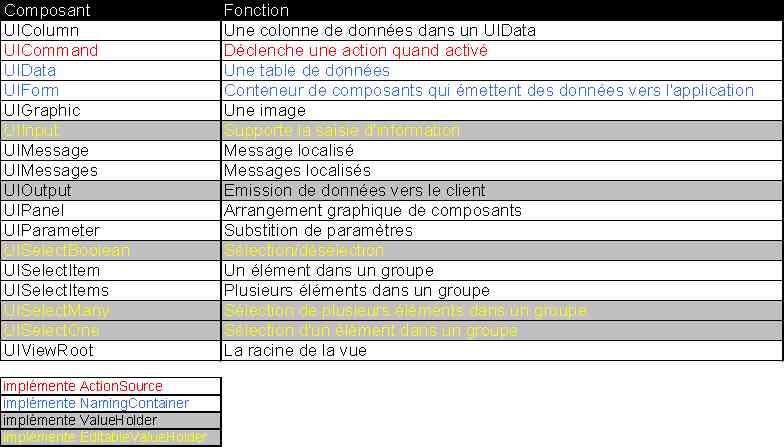
#{client.nom} :

* Invocation du getter sur l'objet
* Renvoie une erreur si la propriété n'existe pas

On peut enchaîner les appels:

#{mesClients[i].adresse.ville}  
#{client.comptes[0].solde}

#### Schéma récapitulatif des composants/tags core



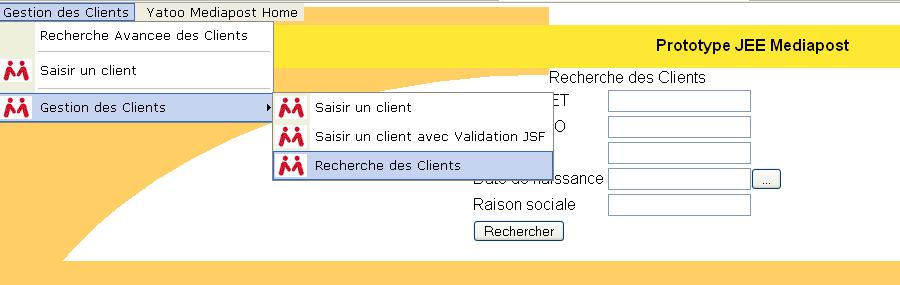
#### Quelques composants

Différents composants externes ont été implémentés dans le prototype. Tomahawk a été choisi comme librairie externe mais les applications pourront rajouter d'autres librairies de composants (tobago, trinidad, etc.).

La configuration Tomahawk a été réalisée dans le prototype. Voici quelques exemples de composants intégrés:

##### TomaHawk : Menu

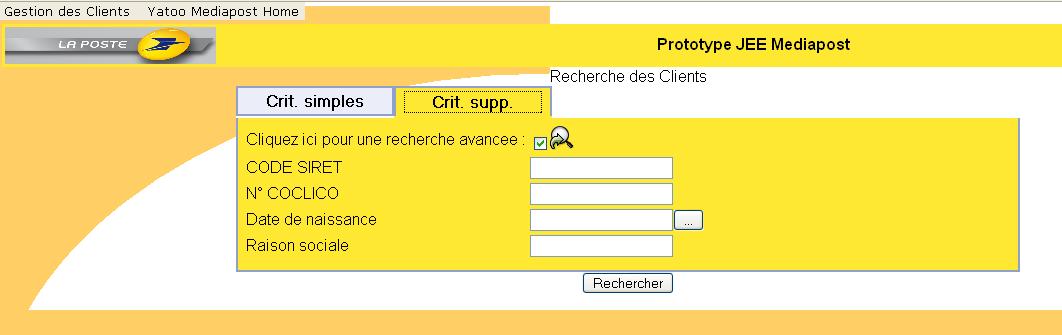
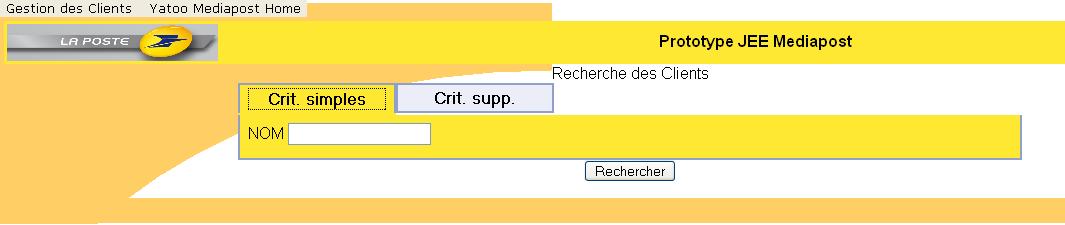
Attention ne jamais placer un menu tomahawk dans un formulaire (balise <h:form/>)



##### TomaHawk : Calendrier



##### TomaHawk : Onglets



## Comment gérer les listes de choix ?

Dans les applications Web, avec JSF, il est fréquent d'utiliser des listes de choix (prenant la forme d'une combo box ou de checkboxes par exemple) dont les éléments sont construits en Java via le BackingBean.  
L'objectif est alors de construire une collection d'objets de type SelectItem.

### Convertir une collection d'objets métiers en une collection de SelectItem JSF

La classe abstraite de BackingBean mise en place dispose de quatre méthodes convertissant une collection d'objets métiers en liste de choix. Deux déclinaisons sont possibles selon que l'on veut avoir une clef correspondant à l'objet ou à une propriété de l'objet.  
Attention, les objets servant de clef doivent implémenter les méthodes **equals** et **hashCode**. Ceci est nécessaire pour pouvoir identifier le choix sélectionnée

|  |  |
| --- | --- |
| **Méthode** | **Description** |
| convertIntoSelectItems(Collection, String) | Utilise une propriété de l'objet comme clé et sa forme chaîne pour valeur. |
| convertIntoSelectItems(Collection, String, String) | Utilise une propriété de l'objet comme clé et une propriété de l'objet pour valeur. |
| convertIntoSelectItemsObject(Collection) | Utilise l'objet comme clé et sa forme chaîne pour valeur. |
| convertIntoSelectItemsObject(Collection, String) | Utilise l'objet comme clé et une propriété de l'objet pour valeur. |

### Construire la vue JSF

L'utilisation de la liste de choix construite se fait à l'aide du tag " f:selectItems ". Attention à prendre garde au type de la propriété liée (via value binding). Elle doit correspondre au type de la clef (la clef est la propriété retournée par SelectItem.getValue()). Si ce n'est pas le cas un convertisseur peut être nécessaire

<h:selectOneMenu id="destinationaccount" value="#{transferBean.destinationAccount}">  
 <f:selectItems value="#{transferBean.accountList}" />  
</h:selectOneMenu>

### Utilisation d'un converter

Le converter a pour objectif de faire le lien entre la représentation issue de la vue et l'objet désigné par le choix. Exemple de code dans le projet BankonetJSF : AccountSelectConverter

## Comment utiliser Tiles ?

La problématique est la maintenabilité des pages JSP.  
Comment construire un site maintenable et évolutif avec un nombre important de pages JSP ?  
L'objectif de la librairie Tiles est d'organiser ses pages autour de composants réutilisables.

Avantages :

* Composants réutilisables et paramétrables
* **Écrits sous forme de patrons de JSP**
* Définitions externalisées en XML
* **Héritage entre composants**

### Exemple de pages jsp sans inclusions:

<html>  
<table border="1" width="70%">  
<tr>  
<td> &nbsp;</td>  
<td> En tete </td>  
</tr>  
<tr>  
<td> Menu </td>  
<td align="center">Contenu de la Page 1 </td>  
</tr>  
<tr>  
<td> &nbsp;</td>  
<td align="center"> Bas de page </td>  
</tr>  
</table>  
</body>  
<html>

<html>  
<table border="1" width="70%">  
<tr>  
<td> &nbsp;</td>  
<td> En tete </td>  
</tr>  
<tr>  
<td> Menu </td>  
<td align="center">Contenu de la Page 2 </td>  
</tr>  
<tr>  
<td> &nbsp;</td>  
<td align="center"> Bas de page </td>  
</tr>  
</table>  
</body>  
<html>

Inconvénients:

* Beaucoup de copier-coller
* Code très peu maintenable

### Exemple de pages jsp avec inclusions:

<html>  
<table border="1" width="70%">  
<tr>  
<td> &nbsp;</td>  
<td> <jsp:include page="entete.jsp" flush="true"/></td>  
</tr>  
<tr>  
<td> <jsp:include page="menu.jsp" flush="true"/> </td>  
<td align="center"> Contenu de la Page 1 </td>  
</tr>  
<tr>  
<td> &nbsp;</td>  
<td align="center"><jsp:include page="basPage.jsp" flush="true"/> </td>  
</tr>  
</table>  
</body>  
<html>

<html>  
<table border="1" width="70%">  
<tr>  
<td> &nbsp;</td>  
<td> <jsp:include page="entete.jsp" flush="true"/></td>  
</tr>  
<tr>  
<td> <jsp:include page="menu.jsp" flush="true"/> </td>  
<td align="center"> Contenu de la Page 2 </td>  
</tr>  
<tr>  
<td> &nbsp;</td>  
<td align="center"><jsp:include page="basPage.jsp" flush="true"/> </td>  
</tr>  
</table>  
</body>  
<html>

Avantages

* Structure plus souple.
* Une modification de l'en-tête ou du pied de page se fait à un seul endroit  
  et est applicable à toutes les pages jsp.

Inconvénients

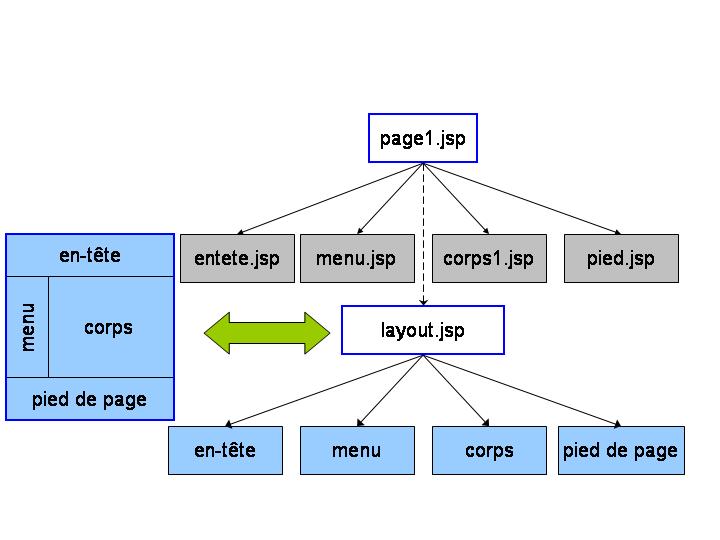
* La structure doit être dupliquée dans chacune des pages.
* Un changement de structure impacte un nombre important de modifications.

### Tiles

Tiles propose ainsi une solution de templating de pages web

La structure des pages est externalisée Un changement de structure sur toutes les pages jsp impacte une seule modification.

Les tiles utilisent les pages incluses. Une modification de l'en-tête ou du pied de page se fait à un seul endroit  
et est applicable à toutes les pages jsp.



### Mise en place des Tiles

Tiles est issu de *Struts*.  
MyFaces Tomahawk fournit une série de composants JSF et propose également une implémentation du  
ViewHandler de la spécification JSF (javax.faces.application.ViewHandler).  
Leur implémentation, JspTilesViewHandlerImpl, permet de gérer les Tiles directement dans le fichier de configuratio de JSF (faces-config.xml).

Il faut donc spécifier ce ViewHandler dans le faces-config.xml:

<faces-config>  
 <application>  
 <view-handler>org.apache.myfaces.tomahawk.application.jsp.JspTilesViewHandlerImpl</view-handler>  
 ...  
 </application>  
 ...  
</faces-config>

L'intégration d'une version de la librairie dans le maven repository a été réalisé sur le prototype.  
Tiles a également été testé sur le prototype : voir ci-dessous

#### Ecrire un Layout

Un layout contient la structure des pages JSP et utilise les balises "tiles" pour prévoir des insertions.

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>  
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>  
<%@taglib prefix="tiles" uri="/WEB-INF/classes/tlds/struts-tiles.tld"%>  
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  
<html>  
<head>  
 <meta http-equiv="Refresh" content="1800;url=timeOutException.htm">   
 <meta content="text/html; charset=ISO-8859-15" http-equiv="content-type">  
 <link rel=stylesheet href="../style/stylemunicipost.css" type="text/css">  
 <title>Application Municipost</title>  
</head>  
<body class="body">  
  
<f:view>  
<f:loadBundle basename="messages" var="messages" />  
  
<tiles:insert attribute="title" flush="false"/>  
<tiles:insert attribute="body" flush="false"/>  
<tiles:insert attribute="footer" flush="false"/>  
</center>  
</f:view>  
  
</body>  
</html>

#### Définition des tiles

Un fichier tiles.xml définit les valeurs à insérer à la place des balises "tiles" du layout.jsp

<!-- Definitions for Tiles documentation -->  
  
<tiles-definitions>  
  
 <!-- ======================================================= -->  
 <!-- Master definition -->   
 <!-- ======================================================= -->  
  
 <!-- Definition de la page mere. -->  
 <definition name="site.mainLayout" path="/tiles-layout/layout.jsp">  
 <put name="title" value="/tiles-component/header.jsp" />  
 <put name="body" value="A DEFINIR" />  
 <put name="footer" value="/tiles-component/menubasedepage.jsp" />  
 </definition>  
  
 <!-- ======================================================= -->  
 <!-- Pages definition -->   
 <!-- ======================================================= -->  
  
 <definition name="/jsps/rech\_client.tiles" extends="site.mainLayout" >  
 <put name="body" value="/jsps/rech\_client.jsp" />  
 </definition>  
 <definition name="/jsps/liste\_client.tiles" extends="site.mainLayout" >  
 <put name="body" value="/jsps/liste\_client.jsp" />  
 </definition>  
 <definition name="/jsps/info\_client.tiles" extends="site.mainLayout" >  
 <put name="body" value="/jsps/info\_client.jsp" />  
 </definition>  
  
</tiles-definitions>

Voici un exemple de règle de navigation JSF:

<navigation-rule>  
 <from-view-id>\*</from-view-id>  
 <navigation-case>  
 <from-outcome>rechercheSuccess</from-outcome>  
 <to-view-id>/jsps/liste\_client.jsp</to-view-id>  
 </navigation-case>  
</navigation-rule>

Automatiquement, grâce à la solution MyFaces Tomahawk, si un tile /jsps/liste\_client.tiles est trouvé  
(ce qui est le cas dans l'exemple du prototype) dans le fichier de définition des tiles (tiles.xml),  
il sera pris en compte et utilisé lors la restitution de la vue.

## Comment utiliser Facelets ?

### Facelets

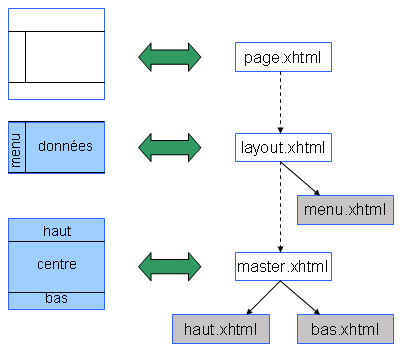
Facelets répond, comme Tiles, à la problématique de constitution de modèles de page (templating) permettant de factoriser du code et de faciliter la maintenance. Les principes de création des vues avec Facelets sont similaires à ceux de Tiles :

* Constitution d’un modèle (voir plusieurs) de page. Il définit les différents éléments constituant la page et leur donne une valeur par défaut. Le modèle peut se voir comme un agrégat de composants.
* Extraction dans des fichiers des différents « composants » de ce modèle. Ex : entête, pieds, menu, …
* Utilisation du modèle pour créer les pages concrètes.

Facelets ajoute certaine facilités liées à JSF :

* Utilisation d’EL à tout endroit de la vue (et pas seulement dans les attributs de balises)
* Factorisation de code par création de taglib purement XML

Contrairement à Tiles, il n’est pas nécessaire d’avoir un fichier XML de plus pour gérer la configuration des modèles. L’utilisation d’un modèle est définie directement dans page. De plus, un modèle (ou une page) peu faire appel à un enchaînement de modèles tout en laissant à la page finale la possibilité de défini la valeur (ou le contenu) de tous les composants définis.



master.xhtml

<?xml version="1.0" encoding="UTF-8"?>  
 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "<http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd>">  
 <html lang="fr" xmlns="<http://www.w3.org/1999/xhtml>"  
 xmlns:ui="<http://java.sun.com/jsf/facelets>"  
 xmlns:h="<http://java.sun.com/jsf/html>"  
 xmlns:f="<http://java.sun.com/jsf/core>"  
 xmlns:c="<http://java.sun.com/jstl/core>"  
 xmlns:t="<http://myfaces.apache.org/tomahawk>">  
 <f:view>  
 <head>  
 <meta content="text/html; charset=ISO-8859-15" http-equiv="content-type" />  
 <t:stylesheet path="/style/stylemunicipost.css" media="all" />  
 <ui:insert name="head" />  
 <title><ui:insert name="page\_title">#{messages.label\_application}</ui:insert></title>  
 </head>  
 <body class="body">  
 <div id="haut"><ui:insert name="haut"> <ui:include src="haut.xhtml" /> </ui:insert> </div>  
 <div id="centre"><ui:insert name="centre"/> </div>  
 <div id="bas">  
 </div><ui:insert name="bas"> <ui:include src="bas.xhtml" /> </ui:insert> </div>  
 </body>  
 </f:view>  
 </html>

layout.xhtml

<?xml version="1.0" encoding="ISO-8859-1"?>  
 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "<http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd>">  
 <jsp:root xmlns:jsp="<http://java.sun.com/JSP/Page>"  
 xmlns:ui="<http://java.sun.com/jsf/facelets>"  
 xmlns:h="<http://java.sun.com/jsf/html>"  
 xmlns:f="<http://java.sun.com/jsf/core>"  
 xmlns:c="<http://java.sun.com/jstl/core>"  
 xmlns:t="<http://myfaces.apache.org/tomahawk>" version="2.0">  
 <html xmlns="<http://www.w3.org/1999/xhtml>" lang="fr">  
 <body>  
 <!-- La déclaration ci-dessus est une astuce pour utiliser l'éditeur de JSP afin d'avoir la complétion syntaxique -->  
 <!-- Rien de ce qui est au-dessus de ui:composition ne sera pris en compte-->  
<ui:composition template="/WEB-INF/layout/master.xhtml"> <!-- Partie principale --><ui:define name="centre"> <h:form id="form\_body">  
  
 <!-- menu --><ui:insert name="menu"> <ui:include src="menu.xhtml" /> </ui:insert> <!-- BODY -->  
 <div id="div\_body">  
 <center><ui:insert name="donnees"> <b>A DEFINIR</b> </ui:insert> </center>  
 </div>  
 </h:form>  
 </ui:define>  
 </ui:composition>  
  
 <!-- Rien de ce qui est au-dessous de ui:composition ne sera pris en compte-->  
 </body>  
 </html>  
 </jsp:root>

page.xhtml

...<ui:composition template="/WEB-INF/layout/layout.xhtml"> <ui:define name="page\_title">#{messages.saisieClient}</ui:define>  
 <ui:define name="donnees">  
 ...  
 <h:commandButton action="#{clientBean.traiterRechercheInit}" value="Rechercher"/>  
 </ui:define>  
 </ui:composition>  
 ...

### Mise en place de Facelets

Facelets est un outil de génération de vues JSF. En premier lieu il faut définir le ViewHandler dédié à ce type de vues dans la configuration JSF (faces-config.xml). En suite, un certain nombre de paramètres sont à préciser dans le fichier web.xml. Une description de la mise en place est disponible ici [Configuration JSF](#scroll-bookmark-227)

Une dépendance supplémentaire est également nécessaire dans votre projet : jsf-facelets-1.1.11.jar

### Format des fichiers contenant les vues

Tous les fichiers utilisés par Facelets sont au format XML. Il est donc impératif de disposer des déclarations d’entités XML relatives au HTML. C’est pourquoi il est adéquat de déclarer ces fichiers comme répondant au format XHTML. En plus de quoi la déclaration de la DTD HTML sera mise en début de fichier.

<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "<http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd>"> <html lang="fr" xmlns="<http://www.w3.org/1999/xhtml>">  
 …

### Création d’un modèle

Pour créer un modèle, il suffit de définir un fichier au format XML (comme vu ci-dessus). Pour définir où doivent être insérés les divers éléments constituant le modèle, il suffit de placer une balise [ui:insert](http://uiinsert) dont l’attribut name est obligatoire. C’est grace à la valeur de cet attribut que seront identifiés les éléments dans les pages utilisant le modèle. Il est conseillé de définir un contenu par défaut à l’intérieur de la balise.

<title><ui:insert name="page\_title">#{messages.label\_application}</ui:insert></title>  
 …  
 <ui:insert name="haut">  
 <ui:include src="haut.xhtml" />  
 </ui:insert>

### Création d’une page

Chaque page va utiliser une modèle de page. Ceci permet uns astuce permettant de faciliter le développement.

#### Intégration de Facelets dans l’IDE

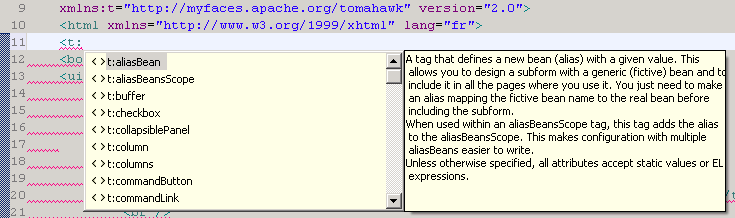
En effet, le support de Facelets dans les IDE n’est pas très développé. Il est possible de s’en passer grâce à la structure d’une page utilisant un modèle à l’aide de la balise [ui:composition](http://uicomposition). En effet seul le contenu de la balise sera exploité par Facelets lors de la compilation de la vue. Il est alors possible de placer ce code dans n’importe quelle structure : celle d’une JSP au format XML par exemple. Pour Eclipse, il suffit alors d’associer l’extension des vues Facelets (.xhtml dans notre cas) à l’éditeur JSP.[Attach:facelets\_eclipse\_contenttype.png](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Facelets?action=upload&upname=facelets_eclipse_contenttype.png)



Puis, de définir les pages dans une structure semblable à celle d’une JSP.

<?xml version="1.0" encoding="UTF-8"?>  
 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "<http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd>">  
 <jsp:root xmlns:jsp="<http://java.sun.com/JSP/Page>"  
 xmlns:ui="<http://java.sun.com/jsf/facelets>"  
 xmlns:h="<http://java.sun.com/jsf/html>"  
 xmlns:f="<http://java.sun.com/jsf/core>"  
 xmlns:c="<http://java.sun.com/jstl/core>"  
 xmlns:s="<http://myfaces.apache.org/sandbox>"  
 xmlns:t="<http://myfaces.apache.org/tomahawk>"   
 xmlns:mdp="<http://fr.mediapost/municipost/facelets/tags>"  
 version="2.0">  
 <html xmlns="<http://www.w3.org/1999/xhtml>" lang="fr">  
 <body>  
 <!-- La déclaration ci-dessus est une astuce pour utiliser l'éditeur de JSP afin d'avoir la complétion syntaxique -->  
  
 <ui:composition template="/WEB-INF/layout/layout.xhtml">  
 …

Cette astuce permet d’avoir la complétion syntaxique sur les tags des différents namespaces utilisés.



#### Utilisation d’un modèle

L’utilisation d’un modèle se fait à l’aide de la balise [ui:composition](http://uicomposition). Le modèle est indiqué à l’aide de l’attribut template qui doit contenir l’URI du fichier contenant le modèle (chemin commençant par /, relatif à la racine de l’application web). Il est utile de remarquer que seule la vue doit être accessible du client final (le navigateur). Le modèle lui peut être placé dans la partie « privées » (WEB-INF).

La balise [ui:composition](http://uicomposition) contient les définitions des divers éléments mis à dispositions par le modèle. Ceux que la vue ne déclare pas seront remplacés par leur valeur par défaut (définie dans le modèle).

Pour définir le contenu d’un élément du modèle il faut utiliser la balise [ui:define](http://uidefine). Son attribut name permet d’identifier l’élément du modèle concerné. A la compilation, tout se passe comme si le contenu de la balise [ui:define](http://uidefine) remplace (dans le modèle) la balise [ui:insert](http://uiinsert) ayant le même nom (cf. attribut name).

### Création d’un composant réutilisable

En plus de l’aspect modèle (templating), Facelets permet également de factoriser du code dans des fichiers au format XML (composition) qui seront utilisable comme des tagslibs.

Il est possible de les utiliser pour factoriser la présentation d’un contenu (titre, note,…) Il est également possible de les utiliser pour factoriser des éléments de définition de l’interface (modèle de tableau, de colonne, d’affichage d’un champ).

Pour construire un tel composant il suffit de

* créer le fichier XML factorisant la portion de code
* créer ou compléter une définition de taglib (en version Facelets).

#### Créer le composant XML

Ce fichier n’est pas différent des vues, il est au format XML et utilise les mêmes tags. Le contenu du composant est défini dans une balise [ui:composition](http://uicomposition). Il est possible de passer des arguments à ce composants, ils seront accessibles par leur nom via EL. Le nom des arguments correspond aux noms des attributs du taglib.

<ui:composition>  
 <table class="titre">  
 <tr>  
 <td class="titre1">#{titre1}</td>  
 <td class="titre2">#{titre2}</td>  
 </tr>  
 </table>  
 <br/>  
 </ui:composition>

#### Définition d’une taglib

La définition du taglib prend la forme d’un fichier XML. Comme pour les autres taglibs (comme les bibliothèques de composants) il faut définir le namespace puis les tags, les tags et les fonctions mis à disposition. Dans notre cas, pour utiliser un code factorisé dans un fichier, il suffit de définir le nom du tag, via la balise tag-name, et le chemin du fichier, via la balise source. Ce chemin est relatif au fichier de déclaration du taglib. Comme pour les fichiers utilisés par les modèles de page, il n’est pas nécessaire qu’ils soient accessibles pour le navigateur.

sampleFaceletsTags.xml

<?xml version="1.0"?>  
 <!DOCTYPE facelet-taglib PUBLIC  
 "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"  
 "facelet-taglib\_1\_0.dtd">  
 <facelet-taglib>  
 <namespace>http://fr.mediapost/municipost/facelets/tags</namespace>  
 <tag>  
 <tag-name>blockTitre</tag-name>  
 <source>tagdef/blockTitre.xhtml</source>  
 </tag>  
 </facelet-taglib>

#### Utilisation du taglib

Avant de pouvoir utiliser le taglib il faut avoir ajouté le fichier de définition dans la liste des taglibs au niveau du fichier web.xml

web.xml

…  
 <context-param>  
 <param-name>facelets.LIBRARIES</param-name>  
 <param-value>/WEB-INF/tld/tomahawk.taglib.xml;/WEB-INF/tld/sampleFaceletsTags.xml</param-value>  
 </context-param>  
 …

Dans la vue devant utiliser ce taglib il faut déclarer le namespace conformément à la définition associée. L’ensemble des tags et des fonctions sont alors disponibles.

<jsp:root  
 xmlns:mdp="<http://fr.mediapost/municipost/facelets/tags>"  
 …

A l’endroit où l’on souhaite l’utiliser il suffit d’y faire appel comme à n’importe quel autre tag. Tout attribut ajouté au tag rendra son contenu accessible dans le composant.

<mdp:blockTitre titre1="&nbsp;" titre2="#{messages.retour\_rechercheclient}"/>

### liens

[documentation de Facelets](http://facelets.dev.java.net/nonav/docs/dev/docbook.html)  
 [pour aller plus loin](http://www-128.ibm.com/developerworks/java/library/j-facelets/)

## Comment gérer le contexte applicatif ?

### Qu'est-ce que le contexte applicatif ?

Dans notre cas, il s'agit du contexte de la partie présentation. La problématique de ce contexte applicatif est de savoir quelles sont les informations que l'on peut y trouver. Techniquement, si l'on adopte un point de vue HTTP plutôt que composants (comme avec JSF), cela revient à réfléchir sur ce qui doit être placé en session ou au niveau de la requête.

### Que mettre dans le contexte applicatif ? Commet veiller à bien le géer (nettoyage)?

On doit mettre principalement des Backing Bean, déclarés dans le fichier faces-config.xml, dans un but de centralisation. On déclare aussi la portée (request, session) dans ce fichier.

Les données utiles (ex. : valeur de champs dans un formulaire) sont déclarés comme propriétés de ces backing bean. Il faut toujours garder à l'esprit que les propriétés d'un backing bean ont la même portée que celui-ci. *Il faut donc éviter de stocker une grande quantité d'informations dans les propriétés d'un backing bean ayant une portée* session, cela afin de ne pas trop charger la session HTTP. Cela s'applique par exemple aux listes de résultats.

Les backing bean vont encapsuler les traitements et des données (méthodes et attributs). Ils seront de portée session dans le majeure partie des cas pour gérer les conversations utilisateurs (enchainements courts de 2 ou 3 écrans ou plus longs de plusieurs écrans). Par contre il faut veiller à nettoyer la session. Par exemple : à chaque choix d'un item d'un menu, les éléments sous-jacents aux écrans du menu précédent peuvent être supprimés de la session.

### Comment accéder au contexte applicatif ?

Comme cela a été mentionné, il est préfèrable d'utiliser le faces-config.xml pour mettre des données en session.

Il est possible d'accéder directement au contexte HTTP en passant par l'ExternalContext :

ExternalContext context = FacesContext.getCurrentInstance().getExternalContext();

Il est alors ensuite possible d'accéder à la session en utilisant directement l’API Servlet (objet HttpSession) pour accéder aux variables de session (ou request selon le cas). Prenons l'exemple d'une récupération d'éléments de session: :

HttpSession session = (HttpSession) context.getSession();  
UserBean userBean = (UserBean) session.getAttribute("userBean");

Ce genre de code est à éviter, car d'une part, il existe un moyen de faire de même via l'API JSF (voir code ci-dessous) et d'autre part, cela dévie du modèle composant de JSF. Néanmoins, pour certaines parties de l'application (authentification, classe de base des backing bean), il est parfois nécessaire d'accéder au contexte du conteneur (de servlet ou de portlet), ce qui nécessite d'utiliser directement l'API Servlet.

Il existe une autre méthode pour récupérer les éléments de la session en utilisant l'API JSF ET en utilisant l'ExternalContext :

UserBean userBean  
 = (UserBean) context.getSessionMap().get("userBean");

Notons qu'il est possible, de la même manière que pour la session, de disposer d'une Map de la request à partir de l'ExternalContext pour récupérer des éléments stockés dans la request :

context.getRequestMap()

Il existe une solution alternative pour récupérer des éléments de différents scopes SANS passer par l'ExternalContext, **cette solution est préconisée et encapsulée dans un helper dédié dans le framework Mediapost (voir section suivante)**. Cette solution est basée sur l’Expression Language JSF. La grande singularité de cette méthode (à la différence de la méthode précédente) réside dans le fait que **le bean est créé s'il n'existe pas**, ce qui est préférable :

UserBean userBean = (UserBean) fc.getApplication().createValueBinding("#{userBean}").getValue(fc);

En précisant un scope:

UserBean userBean = (UserBean) fc.getApplication().createValueBinding("#{sessionScope['userBean']}").getValue(fc);

Les expressions EL JSF sont différentes des expressions EL JSP qui utilisent la notation ${expression}. Les EL JSF (#{...}), contrairement aux EL "standards" JSP (${...}) ne sont pas interprétés par le moteur JSP. Attention JSF 1.2 propose d'utiliser un "Unified EL" avec au choix l'utilisation de #{} ou ${}. Avec JSF 1.2, le code précédent est déprécie, il est conseillé d'utiliser ce code qui de toute façon est destiné à figurer dans le Helper précédemment cité du framework Mediapost:

UserBean userBean = (UserBean) fc.getApplication().getExpressionFactory()  
 .createValueExpression(fc.getELContext(), "#{userBean}", UserBean.class)  
 .getValue(fc.getELContext());

**Le Helper qui est fourni par le framework Mediapost permettrait d'abstraire le code applicatif de cette complexité et surtout permet de l'abstraire de ces éventuels changements entre versions de la spécification JSF!!!**

Il existe une dernière solution consistant à utiliser directement la classe javax.faces.el.VariableResolver, mais nous ne préconisons pas son usage (puisque très similaire à la solution précédente). Avec JSF 1.1:

UserBean userBean = (UserBean) fc.getApplication()  
 .getVariableResolver().resolveVariable(fc, "userBean");

Avec JSF 1.2:

UserBean userBean = (UserBean) fc.getELContext()  
 .getELResolver().getValue(fc.getELContext(), null, "userBean");

### Helper pour la récupération de backing bean

L'idée de ce helper est de faciliter et surtout être une abstraction de la récupération de backing bean dans d'autres backing beans. Son utilisation passe par :

* la déclaration d'une propriété IContextHelper dans le backing bean voulant récupérer d'autres backing
* la déclaration d'un backing bean de type JsfContextHelper (de portée application) dans faces-config.xml
* l'injection du helper dans le backing bean appelant, toujours dans faces-config.xml

On peut enuite récupérer un objet de la façon suivante :

UserBean userBean (UserBean) = contextHelper.getObjectFromSession("userBean");

Pourquoi une propriété dans le backing bean et pas un appel statique ? Si l'on est amené à faire des tests unitaires sur les backing beans, le contexte JSF ne sera pas initialisé et les tests unitaires ne seront pas possibles. Avec une propriété d'instance, il est possible d'injecter au backing une implémentation mock du helper, que l'on peut paramétrer facilement.

Il est cependant toujours possible de faire des appels statiques :

UserBean userBean = (UserBean) JsfContextHelperImpl.getObject("userBean");

Utilisation de l'implémentation mock :

MockContextHelper helper = new MockContextHelper();  
 helper.getRequete().put("myKey", "myValue");  
 // dans le backing beanUserBean userBean (UserBean) = contextHelper.getObjectFromRequest("userBean");

Les différentes classes se trouvent dans le package fr.mediapost.fwk.web.jsf.contexthelper du framework Mediapost.

### Quand implémenter Serializable ?

Les objets qui doivent être mis dans la session HTTP doivent implémenter l'interface Serializable. Cela est aussi utile pour le failover (restaurer les sessions après un crash ou un arrêt du serveur) et le clustering. Donc les classes typiques devant implémenter cette interface sont :

* les objets domaines
* les backing beans de scope session
* les DTO (quand ils sont utilisés)

Un bon moyen de se forcer à implémenter l'interface est de faire hériter tous les objets d'une classe abstraite implémentant Serializable (AbstractDomain, AbstractBackingBean...). Ainsi, lorsqu'on crée une classe concrète, Eclipse affiche un warning si la classe ne déclare pas le champ serialVersionUID. On peut alors remédier au problème via Eclipse (bouton droit sur la ligne concernée dans la vue *Problems*, choisir *Quick Fix* puis *Add generated serial ID*) :

private static final long serialVersionUID = -2806518548060759200L;

## Divers

### Comment gérer la sécurité ?

[Utilisation d'ACEGI](http://sunic.mediapost.fr/dsiwiki/index.php?n=Transverse.Securite)

Il est essentiel de contrôler les url de l'application. Un utilisateur ne doit pas pouvoir par exemple accéder aux écrans de l'application sans s'être auparavant authentifié correctement. Quelles solutions existent ?

#### A la main...

* courant pour l’autorisation (repository des droits et codage de la logique d’autorisation). Une bonne pratique est de capitaliser ce code dans un framework. Choix que nous avons fait pour Mediapost : voir ci-dessous le paragraphe sur les Action Listener
* plus complexe pour l’autorisation (JAAS)
* code de sécurité dilué dans les pages et les objets métier

#### Reposer sur les mécanismes du Serveur d’application:

* propriétaire (même si souvent uniquement par configuration)
* compatible avec JAAS (consultation des informations par l’application)
* ne couvre pas l’autorisation applicative (uniquement ressource) , nécessite du developpement spécifique pour cela (cf. à la main)

#### Reposer sur un produit SSO

* cher ? (sauf si JA-SIG CAS...)
* éternel problème des « agents » à plugger dans la sécurité du serveur d’application (version, ...)
* ne couvre pas l’autorisation applicative (uniquement ressource) , nécessite du developpement spécifique pour cela (cf. à la main)
* extensible à une solution de gestion d’identité packagée (Siteminder, ...)

#### Critères de choix de la solution idéale

* Respect de standards reconnus par la communauté OSS
* Indépendance vis à vis du serveur d’application (mais en conservant la possibilité d’utiliser sa sécurité)
* Gestion des autorisations applicatives (et pas seulement ressource)
* Intrusivité minimale dans le code applicatif
* JSP-friendly pour gestion contextuelle de l’affichage
* Adaptable à un schéma DB de sécurité existant
* Gestion efficiente de caches

#### Choix du framework Mediapost : un Action Listener

Nous avons développé une solution minimale pour gérer la problématique de contrôle des ressources : un listener spécifique permettant d'effectuer ce contrôle, un ActionListener, est positionné sur les ActionEvent permettant d'implémenter des contrôles spécifiques au projet

Voir la classe dans le prototype : fr.mediapost.municipost.web.actionlistener.MunicipostActionListener héritant de fr.mediapost.fwk.web.jsf.actionlistenerMediapostActionListener

Chaque application peut écrire son propre ActionListener héritant de l'Action Listener du framework. Cet Action Listener du framework dispose d'une méthode processAction(ActionEvent actionEvent) appelée avant chaque invocation d'action de type ActionEvent (clic bouton par exemple). Chaque classe fille de cet Action Listener peut surcharger les méthodes suivantes.

/\*\*  
 \* méthode à surcharger dans toute classe actionlistener fille pour décrire le controle propre à l'application  
 \*  
 \*/  
 protected boolean processControle(String beanName, String methodAction, String outcome){  
 return true;  
 }  
  
 /\*\*  
 \* méthode à surcharger dans toute classe actionlistener fille   
 \* pour l'outcome global de renvoi en cas de problème d'autorisation  
 \*  
 \*/  
 protected String getOutcomeKey(){  
 return "erreur\_autorisation";  
 }

Au sein de la méthode processControle, on peut vérifier par exemple si un objet de type contexte contenant un utilisateur existe en HttpSession (voir [Comment gérer le contexte applicatif ?](#scroll-bookmark-238) ).

Si un ActionListener est positionné au sein de l'application, il faut le déclarer dans le fichier de configuration faces-config.xml.

<faces-config>  
 <application>  
 <action-listener>fr.mediapost.municipost.web.actionlistener.MunicipostActionListener</action-listener>  
 </application>  
<faces-config>

**A noter qu'il serait intéressant de mettre en plus un solution plus complète pour gérer l'authentification et l'autorisation applicative de manière unique, centralisée et externalisée du code applicatif. ACEGI est un bon candidat pour ce type de problématique.** Nous préconisons de l'étudier pour le développement du premier projet.

Voici [Une rapide présentation d'ACEGI](#scroll-bookmark-21)

### Comment gérer un événement ?

#### Présentation des événements JSF

JSF se distingue des autres frameworks web par sa possibilité de générer des événements.

Un composant UI peut générer des événements qui seront lancés côté serveur lors de l'analyse de la requête.

Les événements sont alors mis dans une file d'attente et traités à des moments précis par le framework JSF.

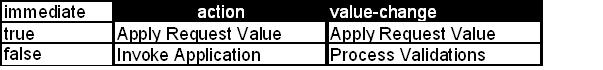
Le modèle événementiel de JSF est basé sur les concepts traditionnels des interfaces graphiques

* Événement
  + Provenance : composant
  + Données
* Listener : callback applicatif

Il existe différents types d'événements :

* Value-change : nouvelle valeur de l’utilisateur sur un UIInput (ex : changement de sélection dans une liste). Evénements générés seulement si les données sont valides
* Action : bouton ou click sur lien (Activation d’un composant qui implémente ActionSource)

Selon que l'attribut immédiat est positionné ou non dans le tag, le traitement de la requête sera différent (voir [Le cycle de vie d'une requête JSF](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.CycleVieRequeteJSF)):



#### Listener

Un listener peut être défini pour un événement soit en écrivant une classe spécifique soit en spécifiant une méthode d'un backing bean

Il peut s'agir d'une classe spécifique. Celle-ci doit implémenter :

* ActionListener
* ValueChangeListener

La déclaration de cette classe est faite alors dans le corps du tag du composant

* f:actionListener
* f:valueChangeListener

Il peut s'agir d'une méthode d'un backing bean. Dans le backing bean, on doit alors avoir :

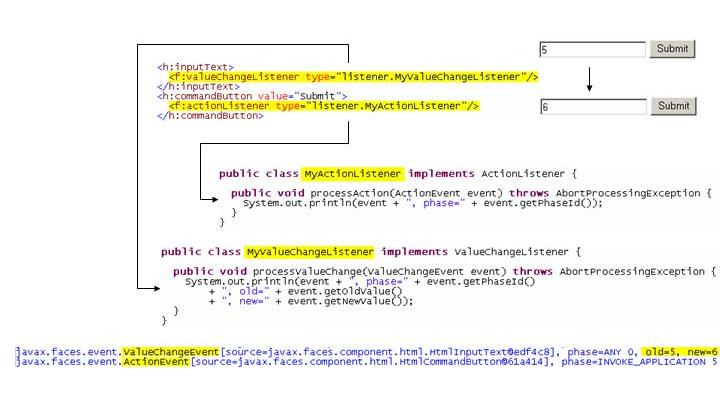
public void <méthode>(ActionEvent event)  
public void <méthode>(ValueChangeEvent event)

Puis la déclaration est effectuée depuis les attributs du composant

actionListener="#{<backing bean>.<méthode>}"  
valueChangeListener="#{<backing bean>.<méthode>}"

Nous préconisons pour des raisons de simplicité les listeners implémentés comme méthodes des backing beans. Néanmoins si un listener doit être réutilisé entre plusieurs backing bean, préférez alors l'implémentation sous forme de classe.

Voici un exemple d'implémentation de listener sous forme de classe:



Voici un exemple d'implémentation de listener sous forme de méthode de backing bean:



**Voir pour une mise en oeuvre pratique de la gestion événementielle [un exemple complet dans le Prototype](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Binding#Prototype)**

### Configuration JSF

Voici les différents éléments du fichier web.xml du prototype, nécessaires pour la plupart à la configuration du framework JSF:

#### Definition de la configuration JSF :

<context-param>  
 <param-name>javax.faces.CONFIG\_FILES</param-name>  
 <param-value>/WEB-INF/classes/faces-config.xml</param-value>  
 </context-param>

#### STATE\_SAVING\_METHOD

Ce paramètre permet de définir sur l'état des composants entre deux requêtes d’un même utilisateur. Il s'agit de l'état des Backing bean, des validateurs, des composants UI, des listeners d'évènement. Il existe deux méthodes :

* Sauvegarde côté serveur, dans la session http
* Sauvegarde côté client, dans un champ caché (input hidden)

Par défaut, utilisez la sauvegarde côté serveur (spécification J2EE). Nous préconisons une sauvegarde côté serveur pour ne pas alourdir les pages. En cas d’utilisation de la persistance sur le client :

* Les composants doivent être sérialisables
* Les objets JSF doivent implémenter StateHolder

<context-param>  
 <param-name>javax.faces.STATE\_SAVING\_METHOD</param-name>  
 <param-value>server</param-value>  
 </context-param>

#### Parametres d'optimisation :

* COMPRESS\_STATE\_IN\_SESSION : permet d'activer/de désativer la compression de l'état côté serveur. A désactiver puisque couteux en performance.
* SERIALIZE\_STATE\_IN\_SESSION : il est conseillé de désactiver la sérialisation de l'état, non nécessaire si l'état est stocké côté serveur. La serialisation et la déserialisation de l'arbre des composants sont très coûteux en terme de performances.
* NUMBER\_OF\_VIEWS\_IN\_SESSION : applicable si l'état est stocké côté serveur. Définit le nombre des dernières vues stockées en session. Peut être augmenté (défaut = 20). Attention, plus ce nombre augmente, plus l'occupation des ressources mémoire sera importante.

<context-param>  
 <param-name>org.apache.myfaces.COMPRESS\_STATE\_IN\_SESSION</param-name>  
 <param-value>false</param-value>  
 </context-param>  
 <context-param>  
 <param-name>org.apache.myfaces.SERIALIZE\_STATE\_IN\_SESSION</param-name>  
 <param-value>false</param-value>  
 </context-param>  
 <context-param>  
 <param-name>org.apache.myfaces.NUMBER\_OF\_VIEWS\_IN\_SESSION</param-name>  
 <param-value>200</param-value>  
 </context-param>

#### Autres paramètres préconisés

Les parametres suivants permettent d'affiner la configuration du framework JSF:

* ALLOW\_JAVASCRIPT : ce paramètre permet d'indiquer à MyFacessi le code javascript peut être autorisé au nivau de la sortie HTML (renderer HTML). Préconisation : à positionner à true (défaut = true)
  + Si le javascript est permis, les tags du type command\_link auront du code javascript qui permettra de soumettre le formulaire correspondant.
  + Si le javascript est interdit, les informations de l'état sauvegardé (voir paramètre STATE\_SAVING\_METHOD) et les paramètres imbriqués seront ajoutés comme des paramètres d'URL.
* DETECT\_JAVASCRIPT : ce paramètre permet d'indiquer à MyFaces si du code javascript peut être autorisé dans le rendu HTML. Par défaut à false
* PRETTY\_HTML : si positionné à true, le code HTML rendu sera formaté pour être "lisible" par un humain. Il contiendra des séparateurs de lignes et des espaces. Préconisation : à positionner à true (défaut = true)
* AUTO\_SCROLL : si positionné à true, une fonction javascript qui sera rendue en sortie sera capable de restaurer le scrolling vertical. Une fonctionnalité idéale si vous avez des pages avec de longues listes et que vous ne voulez pas que la page du naviguateur mette le focus toujours en haut de page, notamment si un lien ou un bouton reste sur la même page. Préconisation : à positionner à true (défaut = false)

<context-param>  
 <param-name>org.apache.myfaces.ALLOW\_JAVASCRIPT</param-name>  
 <param-value>true</param-value>  
 </context-param>  
  
 <context-param>  
 <param-name>org.apache.myfaces.DETECT\_JAVASCRIPT</param-name>  
 <param-value>false</param-value>  
 </context-param>  
  
 <context-param>  
 <param-name>org.apache.myfaces.PRETTY\_HTML</param-name>  
 <param-value>true</param-value>  
 </context-param>  
  
 <context-param>  
 <param-name>org.apache.myfaces.AUTO\_SCROLL</param-name>  
 <param-value>true</param-value>  
 </context-param>

#### Spring

Localisation des fichiers de configuration de Spring et définition du listener :

<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>/WEB-INF/classes/municipostPos-data.xml,/WEB-INF/classes/municipostPos-service.xml,  
 /WEB-INF/classes/municipostPos-db.xml</param-value>  
 </context-param>  
  
 <!-- spring integration -->  
  
 <!-- Permet de repartir le context -definition des beans- en plusieurs fichiers -->  
 <listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
 </listener>

Pour éviter de préciser tous les fichiers, il est aussi possible de préciser un *pattern* :

<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>/WEB-INF/classes/municipostPos\*.xml</param-value>  
</context-param>

Attention, il est nécessaire de définir l'intégration de Spring et JSF au niveau du fichier de configuration de JSF (faces-config.xml).

Cette configuration permet d'accéder à partir du contexte JSF à tout bean défini dans le contexte de Spring:

<application>  
 ...   
 <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-resolver>  
 ...  
 </application>

#### Log4J

Localisation du fichier de configuration de Log4j et définition du listener :

<context-param>  
 <param-name>log4jConfigLocation</param-name>  
 <param-value>/WEB-INF/classes/log4j.properties</param-value>  
 </context-param>  
  
  
 <!-- Log4j listener -->  
 <listener>  
 <listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>  
 </listener>

#### Tiles

Définition du fichier de configuration du ViewHandler : il s'agit du TilesViewHandler

<context-param>  
 <param-name>tiles-definitions</param-name>  
 <param-value>/WEB-INF/classes/tiles.xml</param-value>  
 </context-param>

Attention, il est nécessaire de définir le TilesViewHandler au niveau du fichier de configuration de JSF (faces-config.xml):

<application>  
 ...  
 <view-handler>org.apache.myfaces.tomahawk.application.jsp.JspTilesViewHandlerImpl</view-handler>  
 ...  
 </application>

#### Facelets

L’utilisation de Facelets à la place de Tiles nécessite des paramètres dédiés dans le fichier web.xml :

<!-- Facelets -->  
 <context-param>  
 <param-name>facelets.REFRESH\_PERIOD</param-name>  
 <param-value>3</param-value>  
 </context-param>  
 <context-param>  
 <param-name>facelets.DEVELOPMENT</param-name>  
 <param-value>true</param-value>  
 </context-param>  
 <context-param>  
 <param-name>facelets.LIBRARIES</param-name>  
 <param-value>/WEB-INF/tld/tomahawk.taglib.xml;/WEB-INF/tld/acegijsf.taglib.xml;</param-value>  
 </context-param>  
 <context-param>  
 <param-name>facelets.SKIP\_COMMENTS</param-name>  
 <param-value>true</param-value>  
 </context-param>

* REFRESH\_PERIOD : Période de rafraichissement des définitions de vues après la première compilation. Ceci permet de détecter des changements faits à chaud et de recompiler la vue concernée. Pour désactiver cette fonctionnalité, mettre la valeur -1. Par défaut à 2
* DEVELOPMENT : En cas d’erreur lors du traitement d’une vue, permet d’avoir une description complète de cette erreur et du contexte. Par défaut à false
* LIBRARIES : Liste de chemins séparés par ‘;’ donnant accès aux définitions, pour Facelets, des taglibs.
* SKIP\_COMMENTS : Retire les commentaires présents dans la définition de la vue lors de sa compilation. Attention, les EL présentes dans le commentaire seront malgré tout évaluées. Par défaut à true.

Pour plus de détails vous pouvez consulter [Facelets configuration](http://facelets.dev.java.net/nonav/docs/dev/docbook.html#config)

De plus, les vues en sont plus définies dans des JSP mais dans des fichiers XML :

<context-param>  
 <param-name>javax.faces.DEFAULT\_SUFFIX</param-name>  
 <param-value>.xhtml</param-value>  
 </context-param>

Comme pour Tiles, un ViewHandler dédié doit être mis en place au niveau du fichier de configuration de JSF (faces-config.xml):

<application>  
 ...  
 <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>  
 ...  
</application>

#### Extensions Filter

Quelques composants MyFaces font plus de choses que d'inclure simplement du code HTML dans les pages. Ils ont besoin de ressources supplémentaires comme des scripts javasript, des feuilles de style, images, ...

Ces resources sont incluses dans le jar de MyFaces, mais l'Extensions Filter ajoute en plus le code et l'URL requis pour fournir ces ressources au HTML généré.

Il est donc nécessaire d'ajouter cet Extension Filter au fichier de configuration de JSF (faces-config.xml):

<filter>  
 <filter-name>MyFacesExtensionsFilter</filter-name>  
 <filter-class>org.apache.myfaces.webapp.filter.ExtensionsFilter</filter-class>  
 <init-param>  
 <param-name>maxFileSize</param-name>  
 <param-value>80m</param-value>  
 </init-param>  
 </filter>  
  
 <!-- extension mapping for adding <script/>, <link/>, and other resource tags to JSF-pages -->  
 <filter-mapping>  
 <filter-name>MyFacesExtensionsFilter</filter-name>  
 <url-pattern>\*.jsf</url-pattern>  
 </filter-mapping>  
  
 <!-- extension mapping for serving page-independent resources (javascript, stylesheets, images, etc.) -->  
 <filter-mapping>  
 <filter-name>MyFacesExtensionsFilter</filter-name>  
 <url-pattern>/faces/myFacesExtensionResource/\*</url-pattern>  
 </filter-mapping>

#### Configuration standard de JSF

Il faut spécifier la servlet d'entrée (front controleur) du framework JSF. Dans l'exemple ci-dessous, toutes les URL du type \*.jsf seront traitées par le framework JSF. A noter que la balise load-on-startup dont la valeur est positionnée à 1 demande que la servlet soit chargée dès le démarrage du serveur (et non lors de sa première sollicitation de la servlet: valeur 0). Le nombre entier situé à l'intérieur de ces balises représente l'ordre de chargement. S'il y avait plusieurs servlets dans ce fichier, on pourrait définir la position 2 de chargement pour une autre...

<!-- Faces Servlet -->  
 <servlet>  
 <servlet-name>Faces Servlet</servlet-name>  
 <servlet-class>org.apache.myfaces.webapp.MyFacesServlet</servlet-class>  
 <load-on-startup>1</load-on-startup>  
 </servlet>  
  
  
 <!-- Faces Servlet Mapping -->  
 <servlet-mapping>  
 <servlet-name>Faces Servlet</servlet-name>  
 <url-pattern>\*.jsf</url-pattern>  
 </servlet-mapping>

#### Configuration standard (non liée à JSF)

Page d'acueil de l'application :

<welcome-file-list>  
 <welcome-file>index.jsp</welcome-file>  
 </welcome-file-list>

Timeout de l'HttpSession (en minutes) :

<session-config>  
 <session-timeout>  
 30  
 </session-timeout>  
 </session-config>

Les exceptions Handlers :

<error-page>  
 <exception-type>fr.mediapost.municipost.exceptions.InvalidSubmitException</exception-type>  
 <location>/jsps/invalidSubmitException.jsp</location>  
 </error-page>  
  
 <error-page>  
 <exception-type>fr.mediapost.municipost.exceptions.EndOfHttpSessionException</exception-type>  
 <location>/jsps/timeOutException.jsp</location>  
 </error-page>  
  
 <error-page>  
 <exception-type>java.lang.Exception</exception-type>  
 <location>/jsps/generic\_exception.jsp</location>  
 </error-page>

### Cycle de vie d'une requête JSF

La servlet FacesServlet est le point d'entrée du framework JSF. La réponse JSF est réalisée lors de la dernière phase du traitement de la requête, la phase de rendu. JSF est indépendant du rendu final (Html, Xml, ...). La phase de rendu est réalisée par un 'kit' de rendu (Render Kit). Le kit de rendu par défaut est Html.

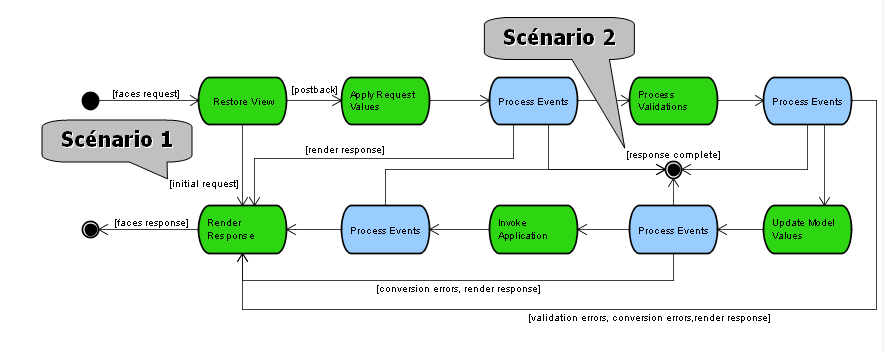
Une réponse Non-JSF est définie comme une page ne contenant pas de composants JSF. Il est possible de sortir du framework JSF à différents endroits du traitement de la requête. La sortie anticipée du traitement se fait par l'appel de FacesContext.responseComplete(). La phase de rendu est alors court-circuitée.

#### Les phases

Pour être flexible, JSF définit des phases précises pour traiter une requête

Les spécifications JSF définissent 6 phases:

* RESTORE\_VIEW : création/restauration de la vue
* APPLY\_REQUEST\_VALUES : récupération des paramètres de requête.
* PROCESS\_VALIDATIONS : validation des données envoyées
* UPDATE\_MODEL\_VALUES : mise à jour du modèle
* INVOKE\_APPLICATION : traitement applicatif
* RENDER\_RESPONSE : génération du flux de réponse et sauvegarde des états des composants UI de la vue retournée



#### Restauration de la vue

Quand une requête JSF est lancée (bouton ou lien), le framework JSF démarre cette phase.

Le but de cette phase est de créer/restaurer la vue JSF de cette requête. L'identifiant JSF de cette requête est extrait. Le FacesContext et cet identifiant permettent de créer / restaurer la vue JSF pour cette requête

Une vue JSF est un arbre de composants UI avec une racine appelée UIViewRoot

* première possibilité: la vue est créée. JSF crée une vue, la sauvegarde et appelle la phase de rendu
* deuxième possibilité: la vue est restaurée. La vue a déjà été appelée, elle est donc récupérée

A l'issue de cette phase, une vue JSF, avec un composant UIViewRoot, a été associée à cette requête

Synthèse de cette phase RESTORE\_VIEW :

* JSF extrait l'identifiant de la requête : /login.jsp par exemple
* Recherche dans le contexte si une vue existe pour cet id
* JSF récupère (ou crée) cette vue, le UIViewRoot racine de cette vue devient le 'viewRoot' courant du contexte
* Les composants enfants sont UIForm,UIInput et UICommand
* Tous les composants UI enfants sont rechargés avec leurs états sauvegardés lors de la requête précédente.

#### Appliquer les Paramètres de la requête

Le but de cette phase est la mise à jour partielle de tous les composants UI de cette vue avec les informations de la requête (paramètres, cookies, headers, ...)

Les composants UI sont appelés récursivement et leur méthode processDecodes() réalise cette mise à jour

Si l’attribut immediate du composant a la valeur true, la validation et le traitement des événements associés sont effectués aussitôt pour le composant.

A l’issue, traitement des événements générés durant cette phase

* FacesContext.renderResponse() : Render Response (appel de la phase de rendu)
* FacesContext.responseComplete() : Stop (sortie du framework JSF)

Synthèse de cette phase APPLY\_REQUEST\_VALUES. Imaginons un formulaire simple avec un champ de type INPUT TEXT et un bouton de validation (INPUT SUBMIT):

* Le composant UIInput délègue l'extraction des paramètres de la requête à son Renderer: uiInput.decode() délègue à renderer.decode()
* Ce Renderer extrait la valeur entrée par l'utilisateur et attribue cette valeur à l'attribut submitValue du composant UIInput
* **Attention : le backingBean n'a pas encore été mis à jour !**
* Le composant UICommand délègue aussi l'extraction de ses paramètres de la requête à son Renderer
* ButtonRenderer extrait la valeur de 'type'. Si la valeur est 'submit', il crée un événement ActionEvent
* Il est ajouté à la file d'attente des événements du composant UI

Au cours des 2 premières phases, les composants peuvent créer des événements:

* Chaque événement est ajouté à la liste d'attente des événements du composant UI source
* Ils sont traités à l'issue de certaines phases
* Chaque composant UI à la source d'un événement doit le diffuser aux listeners intéressés (via sa méthode broadcast())

Fin de la phase APPLY\_REQUEST\_VALUES :

* L'événement ActionEvent du composant UICommand n'est pas traité maintenant car on peut préciser la phase qui doit déclencher l'événement
* L'attribut immediate de UICommand est à false par défaut. Il ne sera pas donc pas traité lors de cette phase

#### Étape de Validation

Le but de cette phase est la validation des données de chaque composant UI.

Voir [La validation](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Validation)

Deux possibilités:

* En cas d'erreur, des messages sont ajoutés par l'appel de FacesContext.addMessage() puis Render Response
* Si les données saisies sont validées, le composant UI est définitivement mis à jour (attribut value)

Synthèse de cette phase PROCESS\_VALIDATIONS. Reprenons l'exemple du formulaire simple avec un champ de type INPUT TEXT et un bouton de validation (INPUT SUBMIT):

* UIInputvalide la donnée saisie
  + Utilise son attribut submitValue (issu de la requête)
  + Applique à cette valeur un éventuel convertisseur
  + Traite l'éventuel attribut required
  + Valide la donnée avec ses éventuels validateurs
* De plus, UIInput peut créer des événements ValueChangedEvent si l'ancienne valeur et la nouvelle sont différentes.
* Tous les listeners seront notifiés à la fin de cette phase

Si, au cours de la phase de validation, des événements critiques ont été envoyés alors deux possibilités:

* renderResponse : Rendu immédiat de la page
* responseComplete : Sortie immédiate du framework JSF

Sinon, la requête a passé avec succès les phases de préparation de JSF

* La hiérarchie des composants UI pour cette requête est prête
* Ces composants UI ont été mis à jour et validés

L'interaction avec notre modèle applicatif peut commencer...

#### Mise à jour du modèle et invocation de l'application

BUT : Mise à jour des « Backing Beans » avec les propriétés des composants UI. Au cours des phases précédentes, seules les valeurs locales des composants UI ont été mises à jour. Rappel : Les propriétés du composant UI peuvent être liées à des « Backing Beans » , point d'entrée vers le modèle applicatif. C'est au cours de cette phase que les attributs du « Backing Bean » sont modifiés. L'attribut value de UIInput a été nullifié, ceci garantit que le « Backing Bean » lié ne sera plus mis à jour par le composant UI

Synthèse de cette phase UPDATE\_MODEL\_VALUES :

* A la fin de cette phase, les « Backing Beans » liés aux composants UI ont été mis à jour
* Notre applicatif peut donc s'appuyer sur ces objets mis à jour par cette requête
* Cette étape permet de notifier à notre système la fin de cette mise à jour
* Les événements standards de JSF ne sont pas traités habituellement lors de cette étape, mais juste après

Après cette phase UPDATE\_MODEL\_VALUES, le modèle est à jour, on traite alors les événements applicatifs. Cette phase ne manipule plus les composants UI, seuls les événements sont traités.

La phase INVOKE APPLICATION permet alors:

* d'exécuter des actions: action, actionListener (method binding)
* de gérer la navigation vers la prochaine vue : action outcome

Parmi les événements de cette phase, traitement fréquent de ActionEvent:

* La page de réponse à afficher est retrouvée à partir des règles de navigation que l'on a définies
* La vue de réponse est créée / retrouvée pour cette page de réponse

Cette vue de réponse est aussi un arbre hiérarchique de composants UI.

L'événement ActionEvent lancé par un UICommand par exemple est géré par le listener par défaut. La sortie (outcome) est une constante : par exemple success. Les règles de navigation indiquent l'id de la vue de réponse (ex: /bienvenue.jsp)

#### Rendu de la page de réponse

BUT : Générer la page qui sera retournée au client:

* La vue de réponse est maintenant définie
* Le framework JSF parcourt cet arbre de composants UI en appelant la méthode encode() de chaque composant
* Chaque composant UI délègue son affichage à son Renderer
* La page de réponse est retournée au client (étape jsp...)
* L'état de la vue de réponse est sauvegardé

Synthèse de cette phase RENDER\_RESPONSE :

* L'id de la vue de réponse est connu (par ex: '/bienvenue.jsp')
* JSF redirige la réponse vers cette vue (par ex: '/bienvenue.jsp')
* La page JSP est traitée normalement et lorsque les Tags JSF sont traités, les composants UI sont appelés
* Chaque Renderer écrit le code HTML de son composant UI
* A la fin du traitement du tag de UIViewRoot, les états des composants UI sont sauvegardés

#### Exemples de scénari

Le schéma montre deux exemples de scénari

* Scénario 1 : initial request
  + Une requête « non » JSF demande une ressource JSF (i.e. : \*.jsf)
  + Transition directe de Restore View vers Render Response
* Scénario 2
  + Une requête JSF génère une réponse « non » JSF
  + Invoquer FacesContext.responseComplete() depuis
    - Apply Request Values
    - Process Validations
    - Update Model Values

### Le binding

Un Java Bean est associé à un ou plusieurs composants contenant de la logique d’interface graphique.

On distingue 2 types d’association :

* Value-binding : association entre la propriété d’un java bean, et
  + la valeur locale d’un composant (ex : formulaire)
  + ou le composant lui-même
* Method-binding : association avec une méthode du bean
  + Validation
  + Conversion
  + Listener

Nous allons nous focaliser sur le **Value-binding**

Ce value-binding est lui-même découpé en deux 2 catégories selon le type de liaison:

* le Value Binding réél :

Liaison de la valeur du composant (lecture/écriture)

* le component Binding :

Liaison du composant lui-même

L' **Expression Language** de JSF (Proche de JSP EL, alignement prévu JSF 1.2/JSP 2.1) permet d'effectuer ce value-binding. Voici un exemple de JSF EL:

#{<expression>}

Nous préconisons d'utiliser le Component Binding pour gérer par des changements d'affichage au niveau des composants, typiquement à la sélection d'un élément dans une liste, des éléments du formulaire peuvent ne plus apparaître. Voir [l'exemple implémenté dans le prototype](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Binding#Prototype)

Voir [Support EL](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Tags#supportel)

#### Value Binding réel

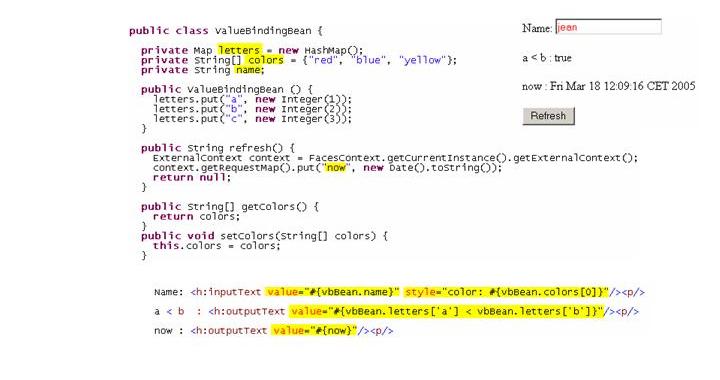
La valeur du composant est liée à une source externe de données (Ex : la propriété d’un objet java). Il s'agit donc de types primitifs, collection (java.util.Collection), tableau ("array"), etc. Une conversion implicite est opérée pour les types standards (ex : int).

Sinon des convertisseurs spécifiques peuvent être utilisés. **L'attribut « value » du tag du composant est utilisé**.

Ex : #{dossier.statut}

Les expressions complexes sont possibles. Elles sont toujours en lecture seule.

Ex : #{caddie.nombreElements > 0}



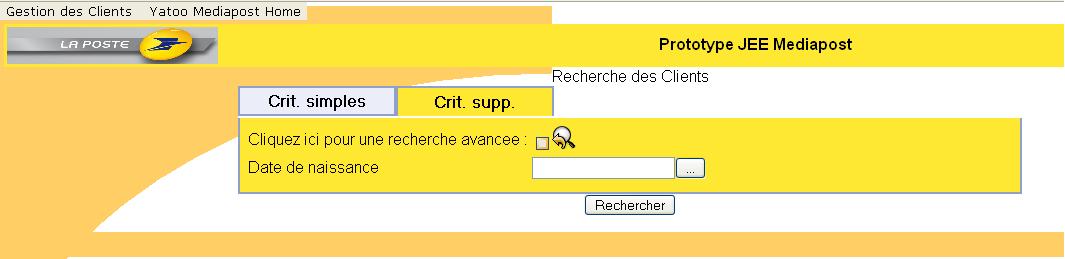
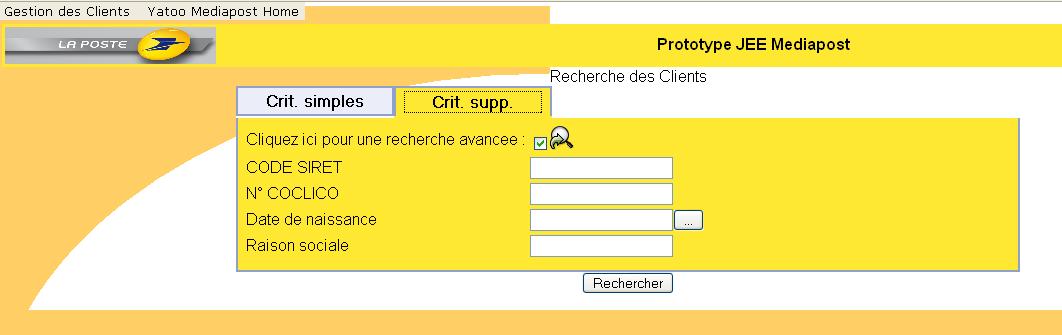
#### Component Binding

Le composant lui-même est lié à la propriété d’un objet  
Il s'agit d'une sous-classe de UIComponent ;  
ce binding permet la manipulation dynamique des propriétés du composant (Ex : attribut rendered)  
Il assure la compatibilité des types  
**L'attribut « binding » du tag du composant est utilisé**



#### Exemple dans le prototype

Nous avons implémenté une recherche simple et une recherche avancée des clients. Selon si une checkbox est cochée ou non pas l'utilisateur, des critères de recherche supplémentaires apparaissent :



Le numéro coclico, le code Siret et la raison sociale sont affichés en fonction de la valeur de l'atribut clientBean.rechercheDetail, grâce au positionnement de l'attribut rendered="#{clientBean.rechercheDetail}"

Voici un extrait de code de la page :

<h:panelGrid columns="2">  
 <h:outputText value="#{messages.choix\_recherche}" />  
 <h:panelGroup>  
 <h:selectBooleanCheckbox value="#{clientBean.rechercheDetail}" onclick="submit()"   
 valueChangeListener="#{clientBean.typeRechercheChanged}" immediate="true"/>   
 <h:graphicImage value="#{clientBean.imagePath}" />  
 </h:panelGroup>  
 ...  
 <h:outputText value="#{messages.client\_siret}" rendered="#{clientBean.rechercheDetail}" />  
 <h:panelGroup rendered="#{clientBean.rechercheDetail}">  
 <h:inputText value="#{clientBean.siret}" id="siret" required="false"/>  
 </h:panelGroup>  
 ...  
 <h:outputText value="#{messages.client\_coclico}" rendered="#{clientBean.rechercheDetail}"/>  
 <h:panelGroup rendered="#{clientBean.rechercheDetail}">  
 <h:inputText value="#{clientBean.coclico}" id="coclico" required="false"/>  
 </h:panelGroup>  
  
 ...  
  
 <h:outputText value="#{messages.client\_raisonsoc}" rendered="#{clientBean.rechercheDetail}"/>  
 <h:panelGroup binding="#{clientBean.panelraison}">  
 <h:inputText value="#{clientBean.raisonsociale}" id="raisonsociale" required="false">  
 <f:validateLength maximum="8" />  
 </h:inputText>  
 </h:panelGroup>  
 ...   
</h:panelGrid>

Un événement est positionné sur la checkbox "recherche avancée" :

valueChangeListener="#{clientBean.typeRechercheChanged}"

Attention ne pas oublier de positionner l'attribut onclick="submit()" et immediate="true". En effet il faut valider le formulaire pour enclencher le traitement de la requête et indiquer par l'attribut immédiate que le traitement de la requête est interrompu (arrêt à l'appel du listener, pas d'invocation de l'application).

Le panel "raison sociale" est bindé par composant, si bien que le listener *typeRechercheChanged* positionné sur l'événement *ValueChangeEvent* permet d'agir sur ce composant, à savoir sur son rendu (le faire apparaître ou disparaître, respectivement panelraison.setRendered(true) et panelraison.setRendered(false)).

Voici le code du backing bean:

public class ClientBean extends MunicipostAbstractBean {  
...  
// Binding par composant  
private UIPanel panelraison;  
private boolean rechercheDetail = true;  
...  
  
 public void typeRechercheChanged(ValueChangeEvent e) {  
 // methode invoquée apres la phase de validation et  
 // avant la phase de mise à jour du modele  
 Boolean value = (Boolean) e.getNewValue();  
 this.rechercheDetail = value.booleanValue();  
 this.processTypeRecherche();  
  
 }  
  
 private void processTypeRecherche() {  
 if(this.rechercheDetail)  
 {   
  
 this.panelraison.setRendered(true);   
 }  
 else  
 {  
 this.panelraison.setRendered(false);  
 }   
 }  
}

### Pattern Open Session In View

Le pattern Open Session In View (OSIV) propose d'avoir une session Hibernate disponible lors du rendu de la vue. L'intérêt est de pouvoir charger à la demande toutes les entités "lazy" (pas encore chargées) d'un graphe d'objet, alors que l'on ne se trouve plus dans la partie métier ou intégration de l'application.

Ce pattern est intéressant pour éviter d'avoir une certaine rigidité dans le chargement du graphe d'objets d'une entité, en fixant par exemple la politique de récupération dans la couche intégration. En contrepartie, il peut aboutir à l'exécution incontrolée d'un grand nombre de requêtes lors du rendu de la vue et donc impacter les performances.

#### Implémentation

Si le pattern OSIV est utilisé, il doit être implémenté d'une façon systématique (transverse) et ne pas avoir d'impact programmatique sur le code. Il existe par exemple une implémentation Spring sous forme d'un filtre de Servlet. Avec JSF, on peut tirer avantage du principe de PhaseListener qui permet d'appeler du code durant le cycle de vie d'une requête JSF. L'utilisation d'un PhaseListener permet de gérer plus finement le cycle de vie de la session Hibernate et la gestion des transactions.

Avec un PhaseListener, on peut par exemple ouvrir une session Hibernate au début de la requête JSF, laisser le code applicatif (phase invoke application) utiliser cette session et gérer les transactions, puis ouvrir une transaction en lecture seule avant la phase de rendu et la fermer juste après cette même phase. C'est ce principe qui est adopté pour le PhaseListener Mediapost.

#### OpenSessionInViewPhaseListener

Une implémentation *mono datasource* a été écrite dans le framework Mediapost :

fr.mediapost.fwk.web.jsf.phaselistener.OpenSessionInViewPhaseListener

Cette implémentation implémente le principe évoqué ci-dessus. Elle ouvre une transaction en lecture seule pour la phase de rendu. La phase invoke application doit gérer elle-même les transactions.

Généralement, la phase invoke application utilise une partie métier (des services) qui est transactionnelle, c'est-à-dire qu'un appel métier depuis un backing bean crée une transaction et la gère complétement. Avec le pattern OSIV, des accès à des objets "lazy" au sein d'un backing bean sont possibles, en revanche ils ne seront pas transactionnels (on ne bénéficiera pas par exemple d'une isolation des transactions).

#### Configuration

*Le* PhaseListener *a des dépendances vers la session factory et le transaction manager*, il faut donc le configurer avec Spring. On peut utiliser pour cela un PhaseListener Spring spécifique. Celui-ci est déclaré dans le fichier de configuration JSF et se chargera de récupérer et d'appeler tous les PhaseListeners déclarés dans le contexte Spring (il s'agit ni plus ni moins qu'un PhaseListener composite car il contient des PhaseListeners et les appelle.

Voici comment déclarer ce PhaseListener (dans le fichier de configuration JSF) :

<faces-config>  
 <lifecycle>  
 <phase-listener>  
 org.springframework.web.jsf.DelegatingPhaseListenerMulticaster  
 </phase-listener>  
 </lifecycle>  
...  
</faces-config>

Puis, il faut déclarer le PhaseListener OSIV et lui injecter ses dépendances :

<!-- phase listener OSIV, chargé via le DelegatingPhaseListenerMulticaster dans faces-config.xml -->  
<bean id="openSessionInViewPhaseListener" class="fr.mediapost.fwk.web.jsf.phaselistener.OpenSessionInViewPhaseListener">  
 <property name="sessionFactory" ref="sessionFactory" />  
 <property name="transactionManager" ref="transactionManager" />  
</bean>

#### TODO : OSIVPhaseListener multi datasources

Pour les cas où une application travaille avec plusieurs base de données et donc plusieurs session factory, le PhaseListener précédent ne fonctionne pas : il faut gérer une session pour chaque session factory.

Idées :

* dépendance vers le PlatformTransactionManager
* dépendances vers une collection de session factories
* toujours une seule transaction
* gestion des ressources (ouverture/fermeture sessions), en les traitant dans une liste

### Une rapide présentation d'ACEGI

[Utilisation d'ACEGI](http://sunic.mediapost.fr/dsiwiki/index.php?n=Transverse.Securite)

#### Problématique

Gérer l'authentification et l'autorisation applicative de manière unique, centralisée et externalisée du code applicatif.

#### Présentation du framework de sécurité ACEGI (licence Apache)

ACEGI est un framework complet de gestion de sécurité applicative.

* Descripteurs pour l'authentification
* Descripteurs pour l'autorisation
* Plusieurs implémentations classiques fournies

Permet d'isoler le code de sécurité du code de l'application (IoC, AOP).

Autosuffisant et utilisant des techniques applicatives (servlet filtre, AOP).

#### Caractéristiques principales

* Solution complète et autonome pour gérer les phases d’authentification et d’autorisation (sans forcément s’appuyer sur la sécurité du Serveur d’application)
* Extrait le code de sécurité de l’application (configuration Spring, AOP, ...)
* Entièrement paramétrable « at run-time » (configuration Spring et éventuellement repository authentifications/autorisations) donc dé-corrélation du cycle de mise en production de l’application
* Fournit de nombreuses implémentations à tous les étages
* Assure la protection de ressources Web
  + Filtres http sur des patterns d’URL
* Assure la protection d’appel de méthodes
  + Intercepteurs AOP
* Assure la protection d’objets passés en paramètre ou reçu en retour lors de l’appel de méthodes

#### L'authentification

ACEGI fourni des interfaces et des implémentations pour la gestion Web des credentials sous forme de

* filtres http implémentant la phase de login (collecte initiale)
  + Form, gérant un formulaire de login (à développer)
  + Basic, popup navigateur transmission en clair
  + Digest, popup navigateur transmission hashée en MD5
  + Anonymous, configuration d’un profil par défaut pour les accès anonymes
  + Remember-me, authentifie l’utilisateur sur la base d’un cookie créé lors d’une précédente visite
* filtres http pour recevoir des credentials « externes » (collecte additionnelle)
  + JA-SIG Central Authentication Service (CAS)
  + SiteMinder
  + Certificat X509
* adaptateurs pour containers externes (sécurité native du serveur application, ...)
  + Fournis pour Catalina/Tomcat, Jetty, JBoss et Resin

ACEGI fournit des interfaces et des implémentations pour l’authentification des credentials (AuthenticationManager)

Plusieurs implémentations de providers fournies:

#### Quelques « caractéristiques sympathiques »

* filtre ConcurrentSessionFilter pour empêcher un utilisateur de se logger plusieurs fois à l’application
* provider RunAsImplAuthenticationProvider pour permettre de substituer une authentification à une autre (ex: appel de Web Services en fournissant une autre authentification, ...)
* filtre ChannelProcessingFilter pour gérer finement les protocoles (http ou https) à imposer suivant les pages de l’application (ex: login forcé en https puis retour à http pour le reste de l’application)
* proxy FilterChainProxy pour chaîner plusieurs filtres ACEGI tout en ayant un seul filtre déclaré dans le web.xml -> performance+
* gestion de cache possible sur certains (tous?) AuthenticationProvider (EH-CACHE au travers de la factory Spring) -> performance+
* gestion d’exceptions avec filtre de routage vers des pages spécifiques et messages (traduisibles) accessibles depuis la request

#### Les autorisations

* Basée sur la notion d’intercepteur
* Permet à la fois de définir
  + les objets à protéger (expression régulière ou pattern type Apache-Ant)
  + les autorisations (≈rôles) nécessaires
  + directement dans l’application context Spring ou dans le code par Jakarta Commons attributes ou Java 5 annotations ou dans tout autre implémentation (à fournir)
* 3 types d’interception possibles:
  + Request Http, par FilterSecurityInterceptor
  + Invocation de méthode, par AOP Alliance MethodSecurityInterceptor (auto proxy dans Spring)
  + JoinPoint, par AspectJSecurityInterceptor

Reste à prendre une décision connaissant

* les autorisations ramenées lors de l’interrogation du provider en phase d’authentification
* la définition de l’objet protégé (URL, méthode, ...) et de sa protection (autorisations nécessaires)

C’est le rôle de l’AccessDecisionManager. L’implémentation fournit le vote démocratique (AccessDecisionVoter) qui peut enchaîner n Voters. 3 implémentations fournies (unanimité, majorité,1 seul Oui suffit).

Les électeurs fournis:

* RoleVoter, se base sur des autorisations de la forme ROLE\_... et vérifie que l’utilisateur possède ce qu’il faut
* ACLVoter, se base sur des autorisations de la forme ACL\_..., gère des Acces Control List sur des objets à protéger (passé en paramètre ou reçu d’un appel de méthode, l’objet doit fournir une méthode permettant de l’identifier)

#### Quelques « caractéristiques sympathiques »

* Bibliothèque de tags pour les jsp
  + authentication (getPrincipal), renvoie les informations de l’utilisateur authentifié
  + authorize (ifAllGranted, ifAnyGranted, ifNotGranted), vérifie les autorisations de l’utilisateur en fonction d’une liste d’autorisations
  + acl (hasPermission), permet de vérifier qu’un utilisateur a les ACL nécessaires pour accéder à une instance d’objet
* AfterInvocationManager, permet de contrôler des objets en retour d’un appel de méthode
  + objet unique: lève une exception si les autorisations ne sont pas suffisantes
  + collection: enlève silencieusement les éléments pour lesquels l’utilisateur n’a pas les autorisations nécessaires

#### Exemple basique

#### Présentation de l'exemple

* Une application jsp avec une page de login simple
* Mise en place de ACEGI
* Sécurisation de la totalité de l’application
* Configuration d’autorisations d’accès suivant les URL
* Utilisation d’un provider d’authentification en mémoire

#### Filtre http

##### Déclaration du filtre ACEGI dans le web.xml (intercepte tout dans ce cas)

<filter>  
 <filter-name>Acegi Filter Chain Proxy</filter-name>  
 <filter-class>org.acegisecurity.util.FilterToBeanProxy</filter-class>  
 <init-param>  
 <param-name>targetClass</param-name>  
 <param-value>org.acegisecurity.util.FilterChainProxy</param-value>  
 </init-param>  
</filter>  
  
<filter-mapping>  
 <filter-name>Acegi Filter Chain Proxy</filter-name>  
 <url-pattern>/\*</url-pattern>  
</filter-mapping>

##### Implémentation du filtre dans l’application context :

<!-- Enchaine les filtres dans l'ordre indiqué -->  
<bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">  
 <property name="filterInvocationDefinitionSource">  
 <value>  
 CONVERT\_URL\_TO\_LOWERCASE\_BEFORE\_COMPARISON  
 PATTERN\_TYPE\_APACHE\_ANT  
 <!-- "httpSessionContextIntegrationFilter" doit être en premier pour recharger le Security contexte acegi -->  
 /\*\*=httpSessionContextIntegrationFilter, authenticationProcessingFilter, anonymousProcessingFilter, exceptionTranslationFilter, filterInvocationInterceptor  
 </value>  
 </property>  
</bean>

##### Implémentation des filtres:

<!-- Recharge le Security context ACEGI -->  
<bean id="httpSessionContextIntegrationFilter" class="org.acegisecurity.context.HttpSessionContextIntegrationFilter">  
</bean>  
<!-- Authentifie l'utilisateur par un form http -->  
<bean id="authenticationProcessingFilter" class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilter">  
 <property name="authenticationManager"><ref bean="authenticationManager"/></property>  
 <property name="authenticationFailureUrl"><value>/mylogin.jsp?login\_error=1</value></property>  
 <property name="defaultTargetUrl"><value>/</value></property>  
 <property name="filterProcessesUrl"><value>/j\_acegi\_security\_check</value></property>  
</bean>  
  
<!-- Permet l'accès anonyme aux ressources publiques: écran login, ... -->  
<bean id="anonymousProcessingFilter" class="org.acegisecurity.providers.anonymous.AnonymousProcessingFilter">  
 <!-- La clé doit être la même que dans le provider ci-après-->  
 <property name="key"><value>toto</value></property>  
 <!-- Attribution du ROLE\_ANONYMOUS -->  
 <property name="userAttribute">   
 <value>anonymousUser,ROLE\_ANONYMOUS</value></property>  
</bean>  
  
<bean id="anonymousAuthenticationProvider" class="org.acegisecurity.providers.anonymous.AnonymousAuthenticationProvider">  
 <property name="key"><value>toto</value></property>  
</bean>  
  
<!-- Gère les exceptions d'authentification et route vers un entryPoint en cas -->  
<bean id="exceptionTranslationFilter" class="org.acegisecurity.ui.ExceptionTranslationFilter">  
 <property name="authenticationEntryPoint"><ref local="authenticationProcessingFilterEntryPoint"/></property>  
</bean>  
<!– Routage vers entryPoint -->  
<bean id="authenticationProcessingFilterEntryPoint" class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilterEntryPoint">  
<!-- Le formulaire doit contenir j\_username et j\_password et submit vers j\_acegi\_security\_check -->  
 <property name="loginFormUrl"><value>/mylogin.jsp</value></property>  
 <property name="forceHttps"><value>false</value></property>  
</bean>  
<!-- Filtre les accès aux URL déclarées et appelle authenticationManager et accessDecisionManager. Les autorisations nécessaires pour accéder à une URL sont listées ici -->  
<bean id="filterInvocationInterceptor" class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">  
 <property name="authenticationManager"><ref bean="authenticationManager"/></property>  
 <property name="accessDecisionManager"><ref local="httpRequestAccessDecisionManager"/></property>  
 <property name="objectDefinitionSource">  
 <value>  
 CONVERT\_URL\_TO\_LOWERCASE\_BEFORE\_COMPARISON  
 PATTERN\_TYPE\_APACHE\_ANT  
 /logoff.jsp=ROLE\_ANONYMOUS,ROLE\_USER  
 /mylogin.jsp\*=ROLE\_ANONYMOUS,ROLE\_USER  
 /\*\*=ROLE\_USER  
 </value>  
 </property>  
</bean>

Le filtre anonymous sert ici à permettre l’accès aux pages ne nécessitant pas une authentification formelle mais restant quand même sécurisées par ACEGI

##### Implémentation du manager authentification

<!-- Cascade les moyens d'authentification configurés tant qu'il n'en trouve pas un qui sache authentifier l'objet Authentication -->  
<bean id="authenticationManager" class="org.acegisecurity.providers.ProviderManager">  
 <property name="providers">  
 <list>  
 <ref local="daoAuthenticationProvider"/>  
 <ref local="anonymousAuthenticationProvider"/>  
 </list>  
 </property>  
</bean>  
<!-- Implémentation provider authentification test en mémoire -->  
<bean id="daoAuthenticationProvider" class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">  
 <property name="userDetailsService"><ref local="inMemoryDaoImpl"/></property>  
</bean>  
<bean id="inMemoryDaoImpl" class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">  
 <property name="userMap">  
 <value>  
 user=password, ROLE\_KINAPASDEDROITS  
 user0=password0, ROLE\_USER,ROLE\_ADMIN  
 ...  
 </value>  
 </property>  
</bean>

##### Implémentation du manager autorisation

<!-- Gère les autorisations d'accès, ici implicitement basé sur les rôles indiqués dans le filterInvocationInterceptor -->  
<bean id="httpRequestAccessDecisionManager" class="org.acegisecurity.vote.AffirmativeBased">  
 <!-- Si tous les voters s'abstiennent alors c'est non -->  
 <property name="allowIfAllAbstainDecisions"><value>false</value></property>  
 <property name="decisionVoters">  
 <list>  
 <ref bean="roleVoter"/>  
 </list>  
 </property>  
</bean>  
<bean id="roleVoter" class="org.acegisecurity.vote.RoleVoter"/>

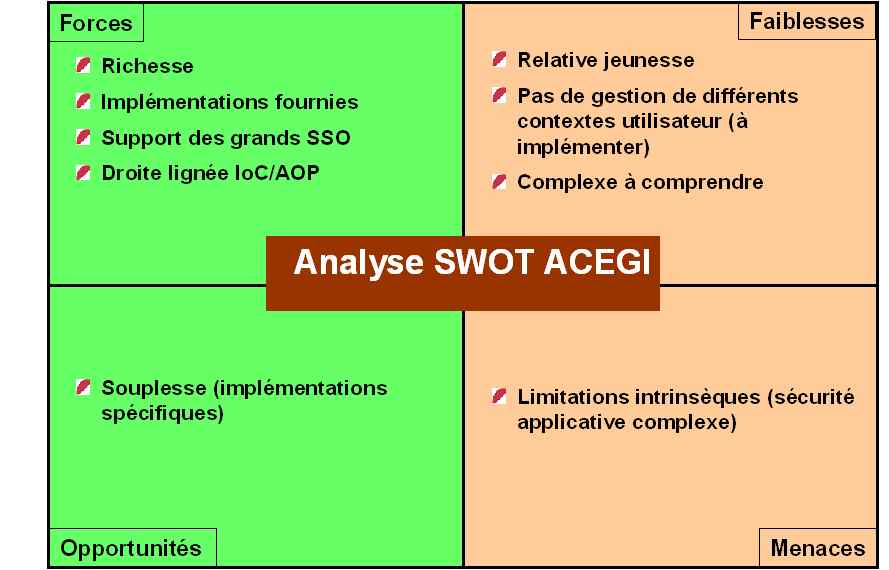
Le RoleVoter prend en compte les autorisations commençant pas ROLE\_...

##### Test et affichage de l’authentification et des autorisations

<%@ page import="org.acegisecurity.context.SecurityContextHolder" %>  
<%@ page import="org.acegisecurity.Authentication" %>  
<%@ page import="org.acegisecurity.GrantedAuthority" %>  
<%@ page import="org.acegisecurity.adapters.AuthByAdapter" %>  
<% Authentication auth = SecurityContextHolder.getContext().getAuthentication();  
if (auth != null) {   
%>  
 Authenticated User: <%= auth.getPrincipal() %><BR><BR>  
 Authentication object holds the following granted authorities:<BR><BR>  
<% GrantedAuthority[] granted = auth.getAuthorities();  
 for (int i = 0; i < granted.length; i++) {   
%>  
<%= granted[i].toString() %> (getAuthority(): <%= granted[i].getAuthority() %>  
)<BR>  
<% }  
} else {   
%>  
 Authentication object is null.<BR>  
 This is an error !!!!<BR><BR>  
<% } %>  
  
<authz:authorize ifAllGranted="ROLE\_ADMIN">  
 Bienvenue Administrateur !  
</authz:authorize>  
  
Bonjour <authz:authentication operation="username"/>

#### Conclusion

* Support actuel assez actif (forums, blog du concepteur Ben Alex, ...) même si peu d’acteurs différents
* Permet d’unifier la gestion de la sécurité et de l’externaliser du code de l’application
* Doc riche mais peu claire, exemples « full-options » lourds à comprendre...
* Implémentations out-of-the-box pour les problématiques courantes
* Bien structuré pour implémenter des spécificités projet (autre repository, autre logique de décision, ...)
* Protection d’objets « amusante », est-ce vraiment utilisable à large échelle ?? (pas de solution élégante trouvée pour l’instant)
* Pb! Ne gère pas de contexte de fonction
  + l’utilisateur a plusieurs fonctions exclusives chacune regroupant des rôles
  + il doit en choisir une en entrant dans l’application et ne conserver que les rôles de cette fonction
  + nécessite des implémentations non fournies et les interfaces ne sont pas prévues pour ce cas précis actuellement
* Pb! Si l’on utilise des composants s’appuyant sur le standard de sécurité Servlet (request.getRemoteUser(), .getUserPrincipal(), .isUserInRole()) il faut mettre en place un filtre spécial pour réinjecter l’authentification ACEGI dans la request (le Principal renvoyé est une implémentation ACEGI offrant des méthodes supplémentaires mais spécifiques...)
* Pas (encore) d’implémentation fournie pour SAML, WS-Security, ... (providers à implémenter)



# Couche Persistance

La couche persistance gère les accès à la source de données (récupération, mise à jour...).

Le framework **Hibernate** est la solution de mapping objet/relationnel (en anglais, ORM pour Object/Relational Mapping) retenue pour cette couche.

## Hibernate

### Evénements et intercepteurs

Hibernate propose 2 façons d'intercepter le processus d'exécution des requêtes, via des intercepteurs et un système de gestion d'événements.

* les intercepteurs permettent d'inspecter et de modifier les propriétés des objets persistants avant que les requêtes ne soient faites en base
* les événements (introduit dans Hibernate 3) sont lancés par l'objet Session et avant les opérations en base. Il est possible d'ajouter des listeners, implémentant certaines interfaces, et donc de réagir aux événements

Le but de ces 2 systèmes est de pouvoir découpler les traitements de persistance communs (récupération, insertion...) d'opérations systématiques.

#### Préconisations

L'architecture événementielle, introduite dans Hibernate 3, permet de répondre à la plupart des besoins, cependant, certaines opérations s'avèrent plus commodes à implémenter avec des intercepteurs. Ainsi, on peut préconiser d'utiliser :

* les intercepteurs pour les opérations nécessitant une inspection, une modification des objets persistants, comme par exemple du tracking (date de création et de modification des objets).
* le système d'événements (i.e. implémenter des listeners et les ajouter à la pile d'Hibernate) pour les opérations nécessitant des vérifications, fonctionnelles ou techniques, susceptibles d'interrompre la chaîne d'exécution d'Hibernate

#### Intercepteurs

Ecrire son propre intercepteur nécessite d'implémenter l'interface Interceptor. Cette interface comportant un grand nombre de méthodes, il est préférable d'hériter de la classe Hibernate EmptyInterceptor. Voici un exemple d'intercepteur mettant à jour la date de création d'un objet lors de son insertion en base :

public class TrackingIntercepteur extends EmptyInterceptor {  
  
  
 @Override  
 public boolean onSave(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types) {   
 if(entity instanceof ITrackable == false) {  
 return false;  
 }   
  
 // pour savoir si l'objet a été modifié   
 boolean changed = false;  
  
 Date currentTime = Calendar.getInstance().getTime();  
  
 ITrackable obj = (ITrackable) entity;  
 // parcours des propriétés  
 for(int i=0,n=propertyNames.length;i<n;i++) {  
 // date de modification  
 if("dateModification".equals(propertyNames[i])) {   
 // non remplie, on met la date courante   
 state[i] = currentTime;  
 // on synchronise l'objet  
 obj.setDateCreation(currentTime);  
 changed = true;   
 }   
 }   
 return changed;  
 }  
}

On ne peut ajouter qu'un intercepteur et cela, de façon programmatique. Il peut cependant être ajouté à plusieurs niveaux : via la configuration (et agit donc sur toutes les Sesssions Hibernate) ou directement sur une Session (et donc n'agit que sur celle-ci) :

// sur la configuration  
new Configuration().setInterceptor( new TrackingIntercepteur() );  
  
// sur une session, à son ouverture  
Session session = sf.openSession( new TrackingIntercepteur() );

Avec le support de base d'Hibernate, l'utilisation d'un intercepteur est relativement limitée, on peut donc préconiser les améliorations suivantes :

* utilisation d'un intercepteur composite : un intercepteur auquel on peut ajouter des intercepteurs. On évite donc d'avoir un seul intercepteur remplissant un grand nombre de fonctions, celles-ci peuvent être déléguées à un ensemble d'intercepteurs. C'est l'intercepteur composite qui est enregistré auprès d'Hibernate et qui délègue le travail à l'ensemble de ces intercepteurs fils.
* configuration déclarative : elle est rendue possible grâce à l'utilisation de Spring.

#### Evénements

Il existe une famille d'événéments, lancés par la Session Hibernate et "autour" (pre-, post-)des opérations base de données. A chaque événement correspond une interface de listener. Il est donc possible d'implémenter ces propres listeners et de les ajouter dans les piles correspondantes.

Pour fixer les idées, voici un exemple de listener (issu de la documentation Hibernate) :

public class MyLoadListener implements LoadEventListener {  
 // this is the single method defined by the LoadEventListener interface  
 public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)  
 throws HibernateException {  
 if ( !MySecurity.isAuthorized( event.getEntityClassName(), event.getEntityId() ) ) {  
 throw MySecurityException("Unauthorized access");  
 }  
 }  
}

L'objectif de ce listener est de filtrer les enregistrements qui peuvent être lus. Il a donc une vocation fonctionnelle.

On peut ajouter le listener de façons déclarative, dans le fichier de configuration d'Hibernate :

<hibernate-configuration>  
 <session-factory>  
 ...  
 <event type="load">  
 <listener class="com.eg.MyLoadListener"/>  
 <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>  
 </event>  
 </session-factory>  
</hibernate-configuration>

ou de façon programmatique :

Configuration cfg = new Configuration();  
LoadEventListener[] stack = { new MyLoadListener(), new DefaultLoadEventListener() };  
cfg.EventListeners().setLoadEventListeners(stack);

Avec la manière déclarative, on ne peut pas appliquer une même instance à plusieurs endroits (une même classe qui implémenterait plusieurs interfaces de listeners). En revanche, cela est possible avec la manière programmatique. L'utilisation de Spring et le référencement de Bean permet de combler ce manque.

##### Listener par défaut

Il est important de noter qu'Hibernate enregistre des listeners par défaut et cela quand aucun listener n'est précisé par le développeur. Ces listeners par défaut ne sont pas enregistrés dès lors que l'on enregistre les siens. Si par exemple, on écrit :

<event type="load">  
 <listener class="com.eg.MyLoadListener"/>  
</event>

et pas :

<event type="load">  
 <listener class="com.eg.MyLoadListener"/>  
 <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>  
</event>

Le listener par défaut effectuant le chargement des objets persistants n'est pas activé et Hibernate se comportera de façon erratique. Il est donc nécessaire, dès lors que l'on rajoute des listeners, *de s'assurer de bien ajouter les listeners par défaut*. Cela rend l'utilisation des listeners assez délicate.

##### Les types de listeners

Pour connaître les types de listeners, il suffit de consulter la documentation API ou la DTD de la configuration Hibernate. Voici les différentes "piles" de listeners :

auto-flush|merge|create|delete|dirty-check|evict|flush|flush-entity|load|load-collection|lock|refresh|replicate  
save-update|save|update|pre-load|pre-update|pre-insert|pre-delete|post-load|post-update|post-insert|post-delete  
post-commit-update|post-commit-insert|post-commit-delete

Il s'agit de toutes les valeurs acceptées par la balise type dans le fichier de configuration :

<hibernate-configuration>  
 <session-factory>  
 ...  
 <event type="load">  
 ...  
 </event>  
 <event type="update">  
 ...  
 </event>  
 <event type="pre-insert">  
 ...  
 </event>  
 <event type="save-update">  
 ...  
 </event>  
 </session-factory>  
</hibernate-configuration>

A partir de ces éléments et de la documentation API, il est aisé de trouver les correspondances entre les classes d'événements et les interfaces de listener.

Il est aussi important de noter que certains événements correspondent directement aux appels de la Session Hibernate. Ainsi, un événement save-update est lancé lors d'un appel de la méthode Session.saveOrUpdate. *Cela ne signifie pas que les listeners* save-update *sont appelés chaque fois qu'un objet est inséré ou mis à jour en base* car

* on peut utiliser la méthode Session.save pour insérer et la méthode Session.update pour mettre à jour un objet persistant
* les appels de Session.save, Session.saveOrUpdate... ne correspondent pas forcément à des opérations en base immédiatement, la synchronisation avec la base étant faite plutôt lors d'un commit de transaction ou d'un flush de Session.

Pour par exemple effectuer des opérations juste avant ou après des ordres SQL, on utilise plutôt les événements pre-insert, pre-update, post-insert, post-update... Les interfaces sont alors de la forme :

boolean onPreInsert(PreInsertEvent event)

Le paramètre de retour booléen précise si la chaîne d'exécution doit être continuée (en retournant false) ou pas (en retournant true).

#### Avertissement

Les intercepteurs et les événements sont des notions avancées et très techniques d'Hibernate. Certaines opérations (ex. : tracking, vérifications diverses) peuvent être implémentées relativement facilement ; néanmoins, des connaissances approfondies des mécaniques de persistance de Hibernate sont rapidement nécessaires pour des traitements plus complexes. Il faut donc utiliser ces outils à bon escient.

### Concurrence d'accès

La politique de gestion de la concurrence choisie pour Mediapost est optimiste puisqu’elle n’est pas basée sur un lock des données. La gestion pessimiste n'est pas abordée puisque, d'une part, elle se justifie la plupart du temps dans des cas très spécifiques et que, d'autre part, elle est généralement déléguée au SGBD via Hibernate.

Hibernate propose un mécanisme automatique pour la gestion optimiste des accès concurrents. Néanmoins, au sein des applications Mediapost, ce n’est pas Hibernate qui va gérer la mise à jour du champ spécifique. La solution est basée sur un champ timestamp au sens Sybase, mis à jour directement par le moteur Sybase. Hibernate va se contenter de lire ce champ.

Le mécanisme proposé repose sur le versionning des objets persistants. Un champ, de type timestamp, est mis à jour automatiquement à chaque UPDATE par le moteur Sybase. Hibernate prend en charge la vérification, c'est-à-dire qu'il s'assure que la version d'un objet sur lequel on effectue la mise à jour n'est pas plus ancienne que l'objet en base. Il est susceptible alors de lancer une StaleObjectException.

#### Mise en place

La mise en place passe par 3 étapes :

* ajouter un champ sur les tables susceptibles d'avoir des accès concurrents, le champ doit être de type timestamp au sens Sybase.
* renseigner le fichier de mapping de la classe (voir ci-dessous)
* ajouter la propriété dans la classe

Le champ doit être un timestamp. Voici le bout de code pour le mapping (celui-ci doit être rajouté après la définition de l'identifiant de la classe) :

<class name="domaine.Personne" table="RS\_PER\_PERSONNE" optimistic-lock="version">  
  
 <id name="id" type="integer">  
 <column name="PER\_ID" not-null="true" />  
 <generator class="increment"/>  
 </id>  
  
 <version type="binary" name="timestamp" generated="always" unsaved-value="undefined">  
 <column name="PER\_TIMESTAMP" />  
 </version>  
  
 <property name="nom" type="string">  
 <column name="PER\_NOM" length="255" not-null="true" />  
 </property>  
 ...   
</class>

A noter:

* generated="always" indique que la valeur de ce champ est générée automatiquement par la base de données
* il est nécessaire de mettre un type hibernate binary pour ce champ version
* cette définition suppose donc qu'une colonne PER\_TIMESTAMP a été créée dans la table avec un type Sybase **timestamp** et que la classe mappée dispose d'une propriété de type byte[] avec les getter/setter correspondants.

Vous pouvez voir un exemple dans le prototype de mise en oeuvre de cette solution de gestion de la concurrence d'accès :

* objet domaine : fr.mediapost.municipost.domaine.hibernate.Personne
* dao : fr.mediapost.municipost.integration.impl.HibernatePersonneDAO
* test unitaire du dao: fr.mediapost.municipost.integration.hibernate.ConcurrenceAccesTest

#### Cas d'utilisation pouvant générer une erreur de versionning

Prenons un formulaire HTML permettant l'édition d'une entité *Client*. Pour faciliter l'explication, nous supposons que nous utilisons un champ de version entier. Imaginons que 2 utilisateurs arrivent en même temps sur le formulaire d'édition du client n°17 et que le numéro de version est à 8. Prenons les étapes suivantes :

* l'utilisateur #1 effectue ses modifications et valide
* l'application prend la main, appelle l'API pour la mise à jour, Hibernate effectue l'incrément. En base de données, le client n°17 a donc maintenant un numéro de version à 9
* l'utilisateur #2 effectue à son tour ses modifications et valide
* l'application prend la main, appelle l'API Hibernate pour la mise à jour. Hibernate effectue la comparaison entre le numéro de version qu'on lui propose (8) et ce qui est en base actuellement (9). Il conclut que l'objet à mettre à jour est plus ancien que la version en base, il lance donc une exception
* l'application gère l'exception et choisit le comportement à adopter (ex. : afficher un message d'erreur à l'utilisateur #2)

Le comportement à adopter lorsqu'un accès concurrent est détecté dépend évidemment de l'aspect fonctionnel.

#### Gestion de l'exception de versionning

Quand Hibernate détecte un accès concurrent il lance une StaleObjectException. Il faut donc que le code applicatif gère ce type d'exception d'une façon ou d'une autre. Voici quelques exemples de traitements possibles :

* le dernier a raison : on force la synchronisation, on risque donc de perdre des informations
* fusion : on récupère la dernière version et on tente de faire une fusion des 2 enregistrements
* message à l'utilisateur : on prévient l'utilisateur que ces modifications n'ont pas été prises en compte et qu'il doit rafraîchir son formulaire

Quand on travaille dans une application en couches, les exceptions de concurrence d'accès sont lancées dans la couche persistance (i.e. contenant les DAO). Comme on l'a précisé, Hibernate lance une exception StaleObjectException, qui est donc une exception propre à ce framework. Pour respecter le principe des couches, il ne faut pas que la gestion de la concurrence, qui peut se passer dans n'importe quelle couche, se base sur ce type d'exception. Une erreur fréquente est de mettre un bloc try/catch sur une StaleObjectException dans la couche coordination (Web, avec une action Struts ou dans un backing-bean JSF). Il faut donc encapsuler les StaleObjectException dans des exceptions applicatives et cela dès la couche de DAO. Cela peut être fait avec des blocs try/catch mais cela devient vite fastidieux. Une autre solution consisterait à intercepter toutes les exceptions de la couche DAO et de les transformer en exception applicative appropriée (possible avec Spring).

#### Gestion des exceptions avec Spring et des intercepteurs AOP

Comme expliqué dans le paragraphe sur la gestion des exceptions de versionning, une bonne pratique consiste à intercepter les exceptions de la partie persistance et à les encapsuler en exceptions applicatives ou génériques. Cela découple donc les couches supérieures des exceptions propres au framework de persistance. Cela se fait en utilisant un proxy pour chacune des classes de DAO et en ajoutant un intercepteur pour les HibernateException. Spring dispose d'une famille d'exceptions génériques et de classes utilitaires pour transformer chaque membre de la famille des HibernateException en un membre de la famille des exceptions de Spring. Ainsi, la gestion des exceptions de la couche persistance dans les couches supérieures peut se faire sur des exceptions indépendantes de l'implémentation de persistance, la transformation se faisant automatiquement et de façon centralisée.

### Hibernate Validator

Hibernate a besoin de meta données pour fonctionner. Les meta données les plus utilisées sont celles définissant le mapping objet relationnel, défini classiquement avec des fichiers XML. Le projet Hibernate Annotations permet de définir ces meta données avec les annotations de Java 5. Il est important de noter que ce projet n'est compatible qu'avec **la version 3.2 de Hibernate**.

Une partie importante du projet Hibernate Annotations concerne la validation des objets de domaine, à base d'annotations. Le framework de validation permet de valider des objets domaine indépendamment de la persistance, cela afin de constituer un référentiel commun de validation, utilisable par toutes les couches de validation (interface graphique, métier, persistance).

Il est possible de faire cohabiter les 2 systèmes de meta données (fichiers XML et annotations). Une préconisation serait donc de conserver toutes les informations de mapping dans des fichiers XML, puisque la plupart des documentations portent sur ce système. Les meta données de validation sont elles définies avec des annotations, directement dans les classes d'entités. Cela évite de surcharger ces dernières.

#### Contraintes

Les meta données de validation de Hibernate Validator, représentées donc sous forme d'annotations, sont appelées des *contraintes*. Elles peuvent accepter des paramètres (une longueur minimum et une longueur maximum pour une chaîne de caractère par exemple). Hibernate Validator dispose d'un certain nombre de contraintes pré-définies (consulter la documentation de référence de Hibernate Annotations pour le détail de chacune des contraintes) :

* @Length(min=, max=)
* @Max(value=)
* @Min(value=)
* @NotNull
* @Past
* @Future
* @Pattern(regex="regexp", flag=)
* @Range(min=, max=)
* @Size(min=, max=)
* @AssertFalse
* @AssertTrue
* @Valid
* @Email

#### Annoter un objet de domaine

On peut annoter un objet de domaine de la façon suivante :

public class Personne implements ITrackable {  
  
 private Integer id;  
 @Length(min=10,max=20)  
 private String nom;  
 @NotNull  
 private String prenom;  
  
 public String getNom() {  
 return nom;  
 }  
  
 public void setNom(String nom) {  
 this.nom = nom;  
 }  
  
 public String getPrenom() {  
 return prenom;  
 }  
  
 public void setPrenom(String prenom) {  
 this.prenom = prenom;  
 }  
}

On définit dans l'exemple ci-dessus les contraintes suivantes :

* la propriété 'nom' doit avoir une longueur comprise entre 10 et 20 caractères
* la propriété 'prenom' ne peut être nulle

On peut aussi imposer plusieurs contraintes à une même propriété :

@Length(min=10,max=20)  
@NotNull  
private String nom;

#### Effectuer une vérification programmatiquement

Hibernate Validator dispose d'une API pour effectuer la vérification des contraintes sur un objet.

// l'objet métier à valider  
Personne personne = new Personne();  
personne.setNom("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");  
  
// création d'un validateur  
ClassValidator personValidator = new ClassValidator( Personne.class );  
// validation de l'objet métier  
InvalidValue[] validationMessages = personValidator.getInvalidValues(personne);  
  
for(InvalidValue invValue : validationMessages) {  
 System.out.println(invValue.getPropertyName() + " : " +invValue.getMessage());  
}

Si on se reporte aux contraintes définies précédemment, on obtient la sortie suivante sur la console à l'exécution de cette routine :

nom : la longueur doit être entre 10 et 20  
prenom : ne peut pas être nul

La validation programmatique est relativement simple à effectuer. Il est ainsi possible d'utiliser Hibernate Validator comme système central de validation dans une application et cela, dans toutes les couches.

#### Listeners de validation

Il est possible d'utiliser le système de validation avec des listeners en pre-insert et pre-update. Ainsi, avant chaque synchronisation entre le graphe d'objets et la base, les entités sont vérifiées par rapport à leur contraintes de validation. Les listeners sont les suivants :

<event type="pre-update">  
 <listener class="org.hibernate.validator.event.ValidatePreUpdateEventListener" />  
</event>  
<event type="pre-insert">  
 <listener class="org.hibernate.validator.event.ValidatePreInsertEventListener" />  
</event>

Ces listeners lèvent une InvalidStateException si la validation échoue. L'exception est typiquement lancée suite à un flush ou à un commit de transaction. Il est donc possible de récupérer, via des intercepteurs AOP et au niveau de la couche DAO, ce genre d'exceptions et les renvoyer aux couches supérieures.

La vérification systèmatique via des listeners peut être utile mais nous préconisons plutôt de faire la validation dans les couches supérieures (métier ou coordination), basée sur les meta données, en utilisant la validation programmatique.

#### Dépendances/interférences entre le système de validation et Hibernate

Le système de validation peut être utilisé de façon complétement indépendante du mapping d'Hibernate. C'est-à-dire que l'on peut annoter des classes non-mappées et appeler l'API de validation sur des instances de ces classes. En revanche, certains attributs de mapping et des contraintes de validation sont communs et on peut observer des interférences.

Prenons par exemple l'attribut de mapping not-null et la contrainte correspondante NotNull. Si on annote une classe avec NotNull, cela s'avère équivalent à utiliser not-null dans un fichier de mapping XML. Donc lors de l'insertion ou de la mise à jour d'un objet en base, le *moteur d'Hibernate* va prendre la main et vérifier l'état des champs annotées avec NotNull. Il s'agira bien du moteur d'Hibernate et pas du système de validation. Cela peut poser problème si l'on s'attend à récupérer des InvalidStateException, par exemple en utilisant les listeners du système de validation.

#### Messages de validation

Les messages de validation par défaut sont contenus dans le package org.hibernate.validator.resources de la bibliothèque Hibernate Annotation. Si les messages ne vous conviennent pas, il est possible de les surcharger en mettant un fichier ValidatorMessages.properties à la racine du classpath (à la racine d'un répertoire source par exemple). On peut post-fixer le nom du fichier de la locale correspondante (ex. : ValidatorMessages\_en.properties). Pour surcharger les messages des contraintes pré-définies, on peut avoir un fichier comme le suivant :

validator.assertFalse=l'assertion a échoué  
validator.assertTrue=l'assertion a échoué  
validator.future=doit être dans le futur  
validator.length=la longueur doit être entre {min} et {max}  
validator.max=doit être inférieur ou égal à {value}  
validator.min=doit être supérieur ou égal à {value}  
validator.notNull=ne peut pas être nul  
validator.past=doit être dans le passé  
validator.pattern=doit suivre "{regex}"  
validator.range=doit être entre {min} et {max}  
validator.size=le nombre d'éléments doit être entre {min} et {max}  
validator.email=Address email mal formée

Ce système permet de centraliser les messages. Si on utilise l'API pour faire les validations programmatiquement, on peut passer directement un ResourceBundle au validateur, afin d'avoir une meilleure granularité sur les messages :

ClassValidator personValidator = new ClassValidator(Personne.class, monResourceBundle);

#### Ecrire ses propres contraintes

Il est possible d'écrire ses propres contraintes afin d'étendre le framework. L'écriture des contraintes se fait en 2 étapes :

* création de l'annotation qui va être ajouté à la classe/propriété/méthode
* création de la classe de validation qui effectuera la validation

Prenons l'exemple d'une contrainte simple : elle vérifie la casse d'une chaîne de caractères. Elle accepte un paramètre qui définit le type de casse à valider (dans notre exemple, tout en majuscule avec 'all' et seule la première lettre en majuscule avec 'ucfirst'). Par défaut (aucun paramètre), la vérification se fait avec 'all'. Définissons d'abord l'annotation :

import java.lang.annotation.Documented;  
import static java.lang.annotation.ElementType.FIELD;  
import static java.lang.annotation.ElementType.METHOD;  
import java.lang.annotation.Retention;  
import static java.lang.annotation.RetentionPolicy.RUNTIME;  
import java.lang.annotation.Target;  
import org.hibernate.validator.ValidatorClass;  
  
@ValidatorClass(CapitalizedValidator.class)  
@Target({METHOD,FIELD})  
@Retention(RUNTIME)  
@Documented  
public @interface Capitalized {  
 /\*\* par défaut, la propriété doit être en majuscule \*/  
 String type() default "all";  
 /\*\* la clé i18n par défaut \*/  
 String message() default "{validator.capitalized}";  
}

On définit ici les caractéristiques classiques d'une annotation (sur quoi elle porte, quand on peut y accéder). On définit aussi son paramètre type, qui précise le type de casse que l'on veut. L'annotation est internationalisée, elle aura la clé validator.capitalized dans le fichier ValidatorMessages.properties. Il faut donc rajouter la ligen suivante dans celui-ci :

validator.capitalized=La casse de cette propriété n'est pas correcte ({type})

On remarque qu'on peut rappeler la valeur des paramètres passés à l'annotation dans le fichier de ressources : notre annotation a un paramètre type et sa valeur peut être récupérée dans le fichier de ressources via le pattern {type}.

On peut passer maintenant à l'écriture de la classe de validation :

import org.hibernate.validator.Validator;  
  
/\*\*  
 \* Vérifie si une chaîne de caractère a une casse correcte.  
 \* @author Arnaud Cogoluegnes  
 \* @version $Id$  
 \*/  
public class CapitalizedValidator implements Validator<Capitalized> {  
  
 /\*\* type de casse \*/  
 private String type;  
  
 /\*\*  
 \* Initialisation liée au framework de validation.  
 \* @param parameters  
 \*/  
 public void initialize(Capitalized parameters) {  
 type = parameters.type();  
 }  
  
  
 /\*\*  
 \* Validation  
 \*/  
 public boolean isValid(Object value) {  
 if (value == null) {  
 return true;  
 }  
  
 // on ne vérifie que les chaînes de caractères  
 if (!(value instanceof String)) {  
 return false;  
 }  
  
 // récupération de la valeur  
 String string = (String) value;  
  
 if ("all".equals(type)) {  
 // casse en majuscule  
 return string.equals(string.toUpperCase());  
 } else if("ucfirst".equals(type)){  
 // seule la première lettre est en majuscule  
 String first = string.substring(0, 1);  
 return first.equals(first.toUpperCase());  
 } else {  
 return true;  
 }  
 }  
}

La classe est plutôt simple à comprendre, elle se contente de vérifier la valeur de la propriété en fonction du type de casse demandée. La méthode isValid doit retourner true s'il n'y pas de problème de validation et false dans le cas contraire.

On peut annoter des méthodes et des champs avec notre contrainte :

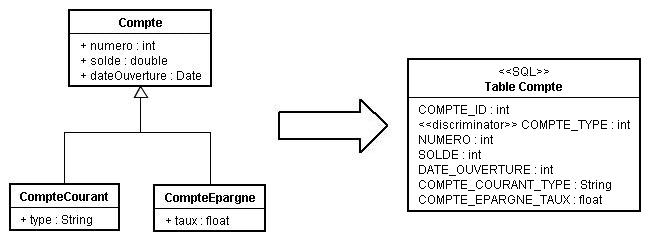
@Capitalized  
private String nom;

### Héritage

Il existe 3 méthodes pour gérer l'héritage avec Hibernate, mais notre étude va se limiter aux 2 méthodes viables.

#### Méthode 1 : une seule table pour le graphe d'héritage

Cette solution offre les meilleures performances et est la plus simple à mettre en place. Elle est présentée dans la figure suivante :



Le mapping Hibernate est alors le suivant :

<hibernate-mapping>  
 <class name="Compte" table="COMPTE" discriminator-value="CA">  
 <id name="id" column="COMPTE\_ID" type="long">  
 <generator class="native"/>  
 </id>  
 <discriminator column="COMPTE\_TYPE" type="string"/>  
 <property name="numero" column="NUMERO"/>  
 ...  
 <subclass name="CompteEpargne" discriminator-value="CE">  
 <property name="taux" column="COMPTE\_EPARGNE\_TAUX"/>  
 </subclass>  
 ...  
 </class>  
</hibernate-mapping>

* La classe mère du graphe d’héritage, Compte, est associée à la table COMPTE
* Afin d’identifier chaque classe, il est nécessaire d’utiliser une colonne spéciale, nommée « Discriminator ». Dans notre exemple, si la valeur de cette colonne est « CE », Hibernate saura qu’il doit créer une instance de CompteEpargne. Il est important de noter que ce « discriminator » est géré automatiquement par Hibernate et qu’il n’est pas nécessaire de déclarer une nouvelle propriété dans la classe persistante.
* Chaque sous-classe est représentée par un élément <subclass>. Attention, ne déclarez surtout pas les colonnes liées aux sous-classes avec une contrainte NOT-NULL. En effet, dans le cas d’un CompteCourant, la colonne COMPTE\_EPARGNE\_TAUX sera nécessairement nulle.

L’élément <subclass> peut lui-même contenir d’autres éléments <subclass>, et ainsi de suite jusqu’à ce que le graphe soit totalement représenté.

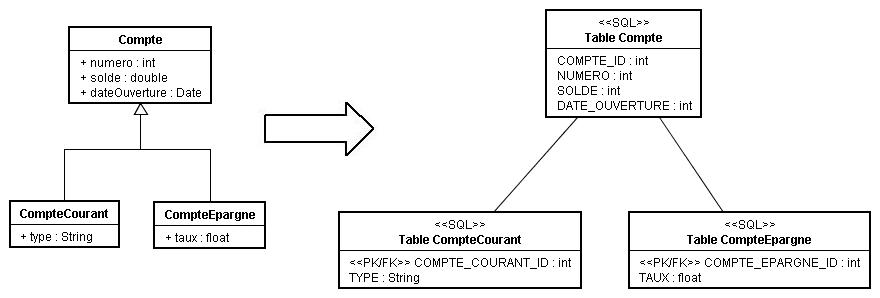
Dans certains cas, une valeur simple sur le discriminateur n'est pas suffisante. Imaginons une classe mère Personne dont le graphe d'héritage est basé sur l'âge. On a donc un ensemble de classe : Enfant, Junior, Adulte... la différenciation se base sur l'âge. Hibernate permet de définir une *formule* qui va être utilisée pour différencier les classes. Voici un exemple :

...  
<discriminator formula="case when PER\_AGE &lt;= 30 then 1 else 2 end" type="integer" />  
...  
<subclass name="fr.mediapost.municipost.domaine.hibernate.Junior" discriminator-value="1" />   
<subclass name="fr.mediapost.municipost.domaine.hibernate.Senior" discriminator-value="2" />

Dans cet exemple, la formule renvoie 1 si le champ PER\_AGE de l'enregistrement est inférieur à 30 et 2 sinon. On voit ensuite qu'une valeur de 1 du discriminateur correspond à la classe Junior et qu'une valeur de 2 correspond à la classe Senior. L'utilisation de *formula* permet notamment de ne pas avoir à modifier une table pour gérer l'héritage.

#### Méthode 2 : une table par sous-classe

Cette solution consiste à déclarer une table pour chacune des sous-classes. On utilise alors une clef étrangère pour représenter l’association vers la table contenant les données de la classe mère :



A la différence de la solution n°1, les tables contiennent des colonnes seulement pour les propriétés non héritées. La clef primaire est également une clef étrangère de la table liée à la superclasse.

L’avantage de cette stratégie est qu’il est facile de faire évoluer le schéma mais qu’il est également possible de déclarer des contraintes d’intégrité.

Le mapping Hibernate est le suivant :

<hibernate-mapping>  
 <class name="Compte" table="COMPTE">  
 <id name="id" column="COMPTE\_ID" type="long">  
 <generator class="native"/>  
 </id>  
 <property name="numero" column="NUMERO"/>  
 ...  
 <joined-subclass name="CompteEpargne" table="COMPTE\_EPARGNE">  
 <key column="COMPTE\_EPARGNE\_ID"/>  
 <property name="taux" column="TAUX"/>  
 </joined-subclass>  
 ...  
 </class>  
</hibernate-mapping>

* L'élément <joined-subclass> permet de mapper une sous-classe à une nouvelle table.
* Une clef primaire est nécessaire pour la table COMPTE\_EPARGNE. Celle-ci doit également être déclarée comme clef étrangère vers la clef primaire de la table COMPTE.

L'élément <joined-subclass> peut lui-même contenir d'autres éléments <joined-subclass> mais pas d'éléments <subclass>.

Etant donné que les requêtes sont très souvent écrites avec des jointures sur plusieurs tables, il est important de noter que cette stratégie offre de mauvaises performances dès lors que la hiérarchie de classes est complexe.

#### Synthèse : quelle stratégie adopter ?

* Si le nombre de données stockées en base est très volumineux, il est préférable d’utiliser la solution n°1. En effet, celle-ci permet d’obtenir les informations nécessaires sans avoir à faire de jointures.
* Si les sous-classes définissent beaucoup de nouvelles propriétés, choisissez la solution n°2. Celle-ci vous évitera d’avoir trop de colonnes non utilisées dans la même table.

### Traces applicatives et profiling des requêtes

Hibernate permet notamment de masquer les API bas-niveau de récupération de données, typiquement tout ce qui est requête SQL. Il arrive parfois que l'on ait besoin de connaître les requêtes effectuées, pour débugguer ou améliorer les performances. Nous allons voir 3 moyens qui permettent d'avoir connaissance de toutes les requêtes SQL qui sont envoyées à la base de données.

#### Avec Hibernate, show\_sql

Une propriété de configuration de Hibernate envoie vers la console toutes les requêtes effectuées par Hibernate. C'est le moyen le plus facile pour avoir une idée des requêtes effectuées. Dans un fichier de configuration Hibernate :

<hibernate-configuration>  
 <session-factory>   
 <property name="show\_sql">true</property>  
  
...  
 </session-factory>   
</hibernate-configuration>

Et dans un fichier de configuration Spring :

<bean id="hibernateProperties"  
 class="org.springframework.beans.factory.config.PropertiesFactoryBean">  
 <property name="properties">  
 <props>  
 ...  
 <prop key="hibernate.show\_sql">true</prop>  
 ...  
 </props>  
 </property>  
</bean>

On a alors une sortie sur la console :

Hibernate: select max(PER\_ID) from RS\_PER\_PERSONNE  
Hibernate: insert into RS\_PER\_PERSONNE (PER\_TIMESTAMP, PER\_NOM, PER\_PRENOM, PER\_AGE, PER\_DATE\_CREATION,  
PER\_DATE\_MODIFICATION, PER\_ID) values (?, ?, ?, ?, ?, ?, ?)

On remarque qu'Hibernate ne renvoie pas les valeurs des paramètres JDBC ('bind parameters'). Ce moyen est le plus facile et rapide à configurer, il permet de se faire une idée du nombre de requêtes et des opérations faites. Si l'on veut vraiment suivre les requêtes générées avec show\_sql, il vaut isoler un cas d'utilisation, typiquement dans un test unitaire, car la sortie peut devenir très rapidement verbeuse. Il est évident que cette option doit être désactivée en production.

#### P6SPY

P6SPY est un driver JDBC qui intercepte les appels JDBC et les retransmet aux véritables drivers. Il nécessite donc un peu de configuration :

* changer le nom du driver à utiliser dans la configuration des datasources (et donc mettre celui de P6SPY)
* paramétrer P6SPY pour utiliser le bon driver et la sortie

Pour l'utiliser il faut donc rajouter le JAR de P6SPY dans le classpath, comme tout autre driver. On configure ensuite les datasources avec le driver P6SPY (exemple dans un datasource avec Spring) :

<property name="driverClass"><value>com.p6spy.engine.spy.P6SpyDriver</value></property>

P6SPY se configure via le fichier spy.properties qui doit être mis à la racine du classpath. Voici des éléments intéressants à configurer :

# le véritable driver à utiliser, avec Sybase par exemple  
realdriver=com.sybase.jdbc2.jdbc.SybDriver  
# la sortie  
# intégration à log4j  
appender=com.p6spy.engine.logging.appender.Log4jLogger  
# vers la console  
appender=com.p6spy.engine.logging.appender.StdoutLogger  
# fichier de P6SPY  
appender=com.p6spy.engine.logging.appender.FileLogger  
logfile=spy.log

En effectuant la sortie vers log4j, on obtient des sorties comme suit :

p6spy - 1157098828014|0|0|statement|insert into RS\_PER\_PERSONNE (PER\_TIMESTAMP, PER\_NOM, PER\_PRENOM, PER\_AGE, PER\_DATE\_CREATION,  
PER\_DATE\_MODIFICATION, PER\_ID) values (?, ?, ?, ?, ?, ?, ?)|insert into RS\_PER\_PERSONNE (PER\_TIMESTAMP, PER\_NOM, PER\_PRENOM,  
PER\_AGE, PER\_DATE\_CREATION, PER\_DATE\_MODIFICATION, PER\_ID) values ('2006-09-01 10:20:27.983', 'Cogo', 'Arno', 27, '', '', 1)

Contrairement à show\_sql, on dispose bien des valeurs des paramètres JDBC liés. P6SPY est plutôt à utiliser en développement, pour débugguer de façon assez fine. Il n'est pas conseillé d'utiliser P6SPY en production.

#### IronTrackSQL

IronTrackSQL est un outil graphique qui utilise P6PSY et log4j pour récupérer à distance les requêtes SQL envoyées à une SGBD. Il permet d'avoir différentes vues des requêtes et fait des statistiques. On a donc une vue beaucoup plus synthétique et exploitable des requêtes. IronTrackSQL est donc plutôt destiné à être utilisé à des fins d'audit et d'optimisation et cela, sur une application installée sur le serveur d'applications.

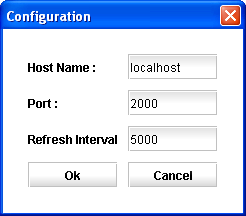
Voici les étapes nécessaires à l'activation de IronTrackSQL dans une application Web :

* ajouter irontracksql.jar, p6psy.jar et log4j.jar dans WEB-INF/lib de l'application
* configurer le fichier spy.properties et le mettre dans WEB-INF\classes
* lancer l'application

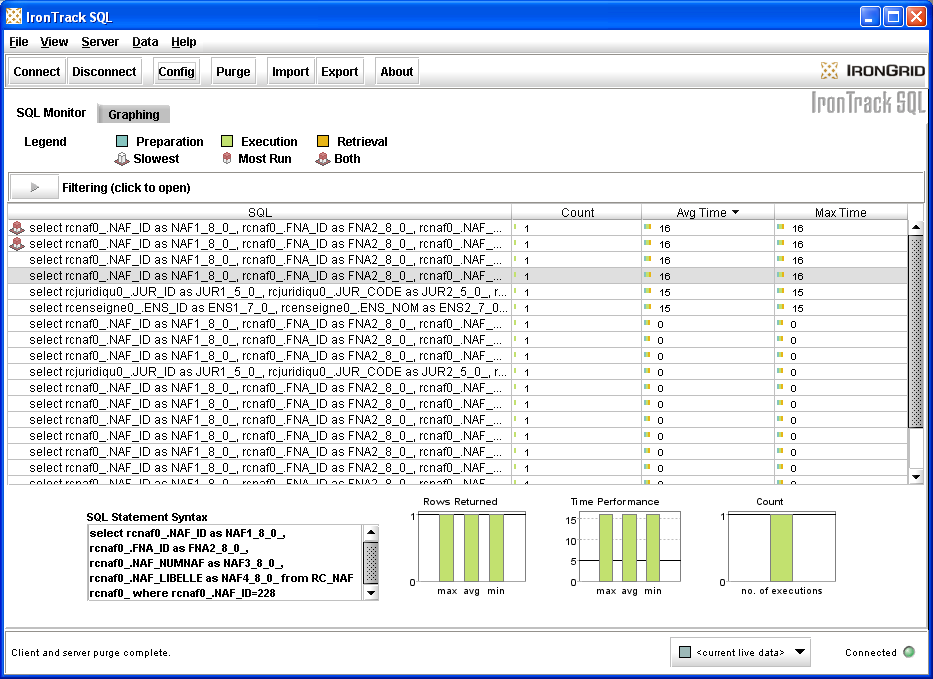
La configuration dans spy.properties se limite à la modification suivante au début du fichier :

#module.log=com.p6spy.engine.logging.P6LogFactory  
#module.outage=com.p6spy.engine.outage.P6OutageFactory  
module.ibeam=com.irongrid.ibeam.server.IBeamFactory  
monitorport=2000

Une fois l'application lancée, toutes les requêtes JDBC peuvent être récupérées par le client de IronTrackSQL. Il suffit de lancer l'application et de configurer le serveur auquel se connecter :



L'interface graphique est ensuite rafraîchie au fur et à mesure que les requêtes sont envoyées :



Parmi les fonctionnalités de IronTrack :

* temps d'exécution moyen, min et max des requêtes
* nombre d'exécutions d'une requête
* affichage ou non des paramètres liés
* nombre de lignes retournées par requête
* répartitions des phases de la requête (préparation, exécution, récupération)

### Utilisation directe de JDBC

Le HQL est très puissant mais il est parfois nécessaire de passer par du JDBC. On peut le faire facilement via l'API Hibernate mais l'on se retrouve finalement à utiliser l'API JDBC, guère commode (exceptions explicites, fermeture des ressources à la main...). Il est donc préférable d'utiliser les classes utilitaires de Spring pour faire du JDBC et cela, au sein de DAO dédiée. Voici un exemple :

import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.jdbc.core.support.JdbcDaoSupport;  
  
import spring.dao.domaine.Utilisateur;  
import spring.dao.integration.IUtilisateurDAO;  
  
public class UtilisateurSpringDAO extends JdbcDaoSupport implements IUtilisateurDAO {  
  
 public void createUtilisateur(Utilisateur utilisateur) {  
  
 JdbcTemplate tpl = getJdbcTemplate();  
 tpl.update(  
 "INSERT INTO utilisateur (login,prenom,nom) VALUES (?,?,?)",  
 new Object [] {utilisateur.getLogin(),utilisateur.getPrenom(),utilisateur.getNom()}  
 );   
 }   
}

Les points importants à noter sont :

* utilisation de la classe de base JdbcDaoSupport de Spring
* utilisation de JdbcTemplate de Spring (exception implicite, pas d'ouverture/fermeture de ressources)
* injection (n'apparaît pas dans le code Java) des dépendances (typiquement un datasource)
* pas de gestion de transaction (le DAO s'insère dans le contexte transactionnel courant). Si on utilise un unique datasource et un HibernateTransactionManager (Spring), on peut utiliser des DAO Hibernate et JDBC dans une même transaction (ceci n'est pas valable avec un autre type de transaction manager, comme JTA par exemple, sauf si on utilise un support pour les transactions distribuées)

Un exemple figure dans le prototype Municipost.

### Méthodes HashCode et Equals sur les objets domaines

Si on utilise Hibernate, il est essentiel de surcharger les méthodse hashCode et equals des objets métiers. Ces méthodes sont notamment utilisées pour l'insertion des objets métiers dans des structures de données (Set, HashMap...) et pour vérifier l'identité des objets.

Par défaut, ces 2 méthodes peuvent se baser sur l'identifiant technique (clé primaire) base de données. Si cela est possible, il est préférable de se baser sur un identifiant fonctionnel (concaténation de plusieurs champs par exemple). L'ouvrage 'Hibernate in action'' décrit clairement tous les subtilités de la surcharge de ces 2 méthodes pour les classes mappées.

La bibliothèque Jakarta Commons Lang dispose d'utilitaires pour implémenter facilement ces 2 méthodes :

public int hashCode() {  
 return new HashCodeBuilder(17, 37).append(getId()).toHashCode();  
}  
  
  
public boolean equals(Object other) {  
 if(other instanceof Personne == false) {  
 return false;  
 }  
 if(this == other) {  
 return true;  
 }  
 Personne otherPersonne = (Personne) other;  
 return new EqualsBuilder().appendSuper(super.equals(otherPersonne))  
 .append(getId(), otherPersonne.getId())  
 .isEquals();   
}

L'utilisation de ces classes utilitaires permet de faciliter et de fiabiliser l'écriture de ce code un peu fastidieux.

## Hibernate FAQ

### Comment récupérer un objet

La méthode session.load() lève une exception runtime s'il n'y a pas de ligne correspondante dans la base de données. Si la classe est mappée avec un proxy, load() retourne juste un proxy non initialisé et n'accède pas à la base de données tant que vous n(invoquez pas une méthode du proxy.  
Ce comportement est très utile s'il est nécessaire de créer une association vers un objet sans réellement la charger à partir de la base de données mais peut être dangereux!!!  
S'il n'est pas certain qu'une ligne correspondante existe, il est préférable d'utiliser la méthode session.get(), laquelle accède à la base de données immédiatement et retourne null s'il n'y a pas de ligne correspondante. Cette fonction retourne toujours une instance de la classe mappée, et non son proxy. Donc, lorsque cet objet est modifié, il faut le synchroniser avec la base de donnée.

### Comment optimiser le chargement des dépendances d'un objet

#### Problème de l'initialisation tardive

Prenons l'exemple de la relation many-to-one. Par défaut avec Hibernate une relation d'un fils vers son père  
(many-to-one) est gérée par proxy. C'est ce qu'on appelle le "chargement par proxy" :  
**Une association vers un seul objet est chargée au moment où un attribut autre que l'identifiant est appelé (via son getter bien sûr) sur l'objet associé.**  
Autrement dit lorsqu'un attribut comme le *nom* du pere (*nom* n'étant pas l'identifiant) est appelé à partir du fils.  
Le comportement peut ne pas être celui attendu par le développeur selon qu'il écrit un service transactionnel ou non.  
Exemple extrait d'un DAO:

...   
 Session session = super.getSession();  
 Object entity = session.get(Fils.class,id);  
 ...

Exemple extrait d'une **méthode transactionnelle** d'un service appelant le DAO:

...   
 Fils monFils = filsDao.find(id)  
 Pere monPere = monFils.getPere(); // OK  
 monPere.getNom(); // OK Puisque la méthode est transactionnelle : la session est ouverte  
 ...

Exemple extrait d'une **méthode non transactionnelle** d'un service :

...   
 monPere.getNom(); // KO Puisque la méthode n'est pas transactionnelle : la session est fermée  
 ...

Le fils dispose ainsi d'un attribut père, lorsqu'un dao récupére ce fils : il s'agit en fait du proxy du père  
(uniquement l'id du père est chargé, et non l'objet complet de Pere)  
Par défaut, Hibernate génère des proxys (au chargement de la sessionfactory) pour toutes les classes persistantes et les utilise pour activer le chargement tardif des associations many-to-one et one-to-one. Si on cherche ainsi à accéder à une donnée du pere autre que l'identifiant, et celà en dehors d'une session  
(autrement dit en dehors d'une méthode transactionnelle), puisque le père n'est en fait pas réellement chargé, une LazyInitialisationException sera levée.

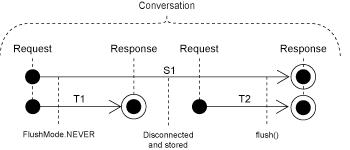
Pour éviter ce problème : plusieurs solutions:

* mettre par défaut au niveau du fichier de mapping un chargement non tardif du Pere (lazy="false").  
  Exemple:

<many-to-one name="pere" class="fr.mediapost.municipost.domaine.MonPere" lazy="false">  
 <column name="PERE\_ID"/>  
 </many-to-one>

**Cette solution va alors dans tous les cas charger le père du fils** ce qui n'est pas optimisé pour tous les cas d'utilisation.  
Typiquement lorsqu'on récupère une liste de fils, dans le seul but d'afficher des données du fils dans un tableau HTML,  
il ne faut pas récupérer toutes ses dépendances many-to-one ou one-to-one si elles ne sont pas requises pour l'affichage.

* Une autre option est de laisser la session ouverte jusqu'à ce que toutes les dépendances  
  (les collections en lien one-to-many et tous les proxys en lien many-to-one) aient été chargées.  
  Dans les architectures à plusieurs couches applicatives, il peut devenir problématique de garantir  
  que la Session soit ouverte lors de l'initialisation. Il y a plusieurs moyens de traiter ce problème :
  + pattern **Open Session in View** : s'appuie sur un filtre de servlet. Ce filtre ferme la Session uniquement  
    lorsque la requête a été entièrement traitée, lorsque le rendu de la vue est fini. Avec cette solution  
    la gestion transactionnelle doit donc être gérée au niveau de ce filtre ; si une exception est levée  
    il est donc nécessaire de rollbacker correctement la transaction. Ce pattern n'est pas compatible avec une architecture  
    cible multi-couche au sein de laquelle la couche service/métier est autonome et gère elle-même le transactionnel  
    (via spring de manière déclarative en l'occurence). Dans ce type d'architecture, cette couche service  
    doit "préparer" toutes les données déjà initialisées qui seront nécessaires à la couche présentation pour un cas d'utilisation donné.
  + pattern **Session-per-conversation** - Une Session Hibernate peut être déconnectée de la couche JDBC  
    après un service métier (apres le commit) puis reconnectée lors d'une nouvelle request  
    (voir schéma ci-dessous issu de la documentation Hibernate). Entre deux request  
    la session est stockée en HttpSession. Cette solution n'est pas satisfaisante si la session hibernate  
    manage un nombre important d'objet, ce qui peut être le cas au sein d'une application d'entreprise.  
    L'objectif est toujours d'alléger le contenu d'une session HTTP.



* **Au vue de ces deux précédents points, nous préférerons cette dernière solution :**  
  Laisser le comportement pas défaut (chargement par proxy) mais appeler un service particulier *transactionnel* qui

permettra au fils, dans le cas par exemple où on veut accéder à son détail, de récupérer toutes ses dépendances.Les méthodes statiques Hibernate.initialize() et Hibernate.isInitialized() fournissent

à l'application un moyen de travailler avec des proxys ou des collections initialisés.

Hibernate.initialize(monfils)

forcera l'initialisation d'un proxy de monfils, à condition que sa Session soit ouverte.

Hibernate.initialize(monfils.getPere())

a le même effet sur la dépendance Pere.**Pour les relations père vers fils (one-to-many)**, la méthode Hibernate.initialize() doit être appelée **si nécessaire** pour chaque collection  
(une collection de fils en l'occurence par exemple) avant la fermeture de la session.  
Il est également possible de récupérer les collections de manière agressive à l'aide :  
**Pour les relations fils vers père (many-to-one)**, un service a été écrit dans le prototype.  
L'interface du service est : fr.mediapost.municipost.service.utils.IGenericDomainLoaderL'implémentation : fr.mediapost.municipost.service.utils.impl.GenericDomainLoaderCe service dispose d'une méthode : public Object getAsObject(Object obj, String value, List dependencies)  
Il permet de charger un object avec ses relations many-to-one spécifiées dans la liste dependenciesManyToOne.Ces dépendances sont à rajouter sous forme de nom d'objet (type String) spécifiant la dépendance.Le service retourne alors automatiquement l'objet avec toutes ses dépendances many-to-one.Le service n'est pas récursif et ne va pas charger les propres relations many-to-one des dépendances de l'objetL'utilisation de ce loader ne permet pas de faire des jointures, ainsi si la liste des dépendances dépasse la taille de 5 éléments, nous conseillons de passer par une jointure (HQL ou criteria) n'impliquant pas plusieurs requêtes SQL (préférable du point de vue des performances).Voir également le test unitaire de ce service : fr.mediapost.municipost.integration.hibernate.LoaderTest

#### Batch-fetching

Pour améliorer les performances, Hibernate peut utiliser le batch fetching. Le batch fetching est une optimisation liée à la stratégie de chargement tardif. Il peut être utilisé au niveau de la classe ou au niveau de la collection.

L’exemple donné dans la documentation Hibernate du batch fetching est simple à comprendre. En voici une explication synthétique sur un exemple simple d’une classe Fils disposant d’un lien vers son Pere (un Pere pouvant avoir plusieurs Fils, un Fils n’ayant qu’un seul Pere). Une session Hibernate peut être chargée avec 30 instances de la classe Fils, chaque Fils ayant une référence à son père. La classe Pere est mappée avec un proxy (relation many-to-one, lazy="proxy"). Lorsque l’on itère sur tous les Fils et que l’on cherche à accéder aux attributs du Pere, Hibernate va exécuter par défaut 30 SELECT, pour charger les Pere (initialiser le proxy). Vous pouvez paramétrer ce comportement en spécifiant un attribut batch-size (taille de batch) dans le mapping de Pere:

<class name="Pere" batch-size="10">...</class>

Hibernate exécutera désormais trois requêtes, en chargeant 3 fois 10 entités. Cette optimisation dépend du nombre de proxys non initialisés dans une Session donnée.

Vous pouvez aussi activer le batch fetching pour les collections. Par exemple, imaginons que chaque Pere ait une collection chargée tardivement de Fils, et que 10 Peres soient actuellement chargés dans la Session. Le fait d’itérer sur tous les Peres et d’accéder aux entités de chaque collection de Fils (par exemple afficher le nom de chaque Fils), générera 10 SELECTs du type « select \* from Fils where pere\_id= ? ». Si vous activez le batch fetching pour la collection Fils dans le mapping de Pere, Hibernate pourra précharger les collections :

<class name="Pere">  
 <set name="fils" batch-size="3">  
 ...  
 </set>  
</class>

Avec une taille de batch (batch-size) de 3, Hibernate chargera respectivement 3, 3, 3, et 1 collections de Fils en 4 SELECTs. La valeur de l'attribut batch-size défini combien de collections peuvent être chargées en un seul select (donc son intérêt dépend du nombre de collections non initialisées dans une Session particulière), ce paramètre n’a pas d’influence sur le chargement des éléments à l’intérieur de la collection. Le batch fetching de collections est particulièrement utile si vous avez des arborescences récursives d'éléments (un Pere dispose d’une collection de Fils, chaque Fils dispose d’une collection d’Animaux de compagnie, etc.)

#### Syndrome des « N+1 » select

Ce problème est lié au cas de figure du chargement tardif des collections. En pratique dès que l’application va accéder aux éléments de la liste chargée tardivement, Hibernate va générer un appel base de données pour chaque élément : on obtient donc N + 1 (1 pour l’initialisation de l’objet initial) appels vers la base de données. Ceci est d’autant plus sensible que cette complexité peut être multipliée par le nombre d’utilisateurs connectés.

S'il peut paraître évident, ce problème est souvent masqué par la complexité globale de l’application. Seule une analyse minutieuse des traces SQL peut y remédier. Un des solutions à ce problème que nous préconisons est le « batch fetching ». Il suffit, pour cela, de déclarer l’attribut fetch=join dans vos collections :

<set name="fils" fetch="join">  
<key column="CLIENT\_ID"/>  
 <one-to-many class="Fils"/>  
</set>

Ceci s’applique alors pour les méthodes get() et load(), la récupération implicite lorsqu’on navigue à travers une association, les requêtes de type Criteria ou HQL.

Si vous ne désirez pas utiliser ce mode pour les méthodes get() ou load(), il faut alors passer par une requête de type Criteria :

Pere pere= (Pere)session.createCriteria(Pere.class)  
 .setFetchMode("fils",FetchMode.JOIN)  
 .uniqueResult() ;

Vous pouvez alors choisir le moment où vous exécutez un chargement par batch.

### Suppression des éléments d’une collection

Parfois, effacer les éléments d'une collection un par un peut être extrêmement inefficace, or il s’agit du fonctionnement par défaut d’Hibernate. Parfois il peut être souhaitable, pour des raisons de performances d’effacer la totalité d’une collection (en un DELETE SQL). Hibernate ne fonctionne pas, par défaut, en supprimant en une fois tous les éléments d’une collection (pour préserver le fonctionnement de triggers base de données)

Il est possible de forcer ce comportement en libérant (c'est-à-dire en déréférençant) la collection initiale et en retournant une collection nouvellement instanciée avec aucun élément. Ceci peut être très pratique et très puissant de temps en temps.

### Doit-on utiliser HibernateTemplate ?

Spring fournit une classe HibernateTemplate qui fait office de "helper" pour utiliser Hibernate. Les DAO héritant de HibernateDaoSupport peuvent récupérer et utiliser un HibernateTemplate via la méthode getHibernateTemplate(). Un des principaux intérêts de cette classe est qu'elle ne retourne pas d'exceptions Hibernate, celles-ci sont directement converties en exceptions génériques de Spring. Pour ce qui est de l'API de HibernateTemplate, elle reste très proche de l'API de la Session Hibernate.

**L'utilisation de** HibernateTemplate **n'a donc pas une grande valeur ajoutée, et ce même avec la version 2.0 de Spring, surtout si un mécanisme de conversion des exceptions est mis en place au-dessus de la couche DAO et que l'on utilise le support DAO Hibernate de Spring.** On peut directement utiliser l'API Hibernate en récupérant la session avec getSession(false) dans un DAO Hibernate et bénéficier de toute la documentation et de tous les exemples concernant cette API.

# Transverse

## Gestion des exceptions

### Généralités

Le système des exceptions pour gérer les erreurs se situe au niveau du langage de programmation et parfois même au niveau du système d'exploitation. Une exception est un objet qui est "émis" depuis l'endroit où l'erreur est apparue et peut être interceptée par un gestionnaire d'exception conçu pour gérer ce type particulier d'erreur.

C'est comme si la gestion des exceptions était un chemin d'exécution parallèle à suivre quand les choses se gâtent. Et parce qu'elle utilise un chemin d'exécution séparé, elle n'interfère pas avec le code s'exécutant normalement. Cela rend le code plus simple à écrire car on n'a pas à vérifier constamment si des erreurs sont survenues.

De plus, une exception émise n'est pas comme une valeur de retour d'une fonction qui peut être ignorée. Une exception ne peut pas être ignorée, on a donc l'assurance qu'elle sera traitée quelque part. Enfin, les exceptions permettent de revenir d'une mauvaise situation assez facilement. Plutôt que terminer un programme, il est souvent possible de remettre les choses en place et de restaurer son exécution, ce qui produit des programmes plus robustes.

Les exceptions sont des instances de sous-classes des classes :

* java.lang.Error pour des erreurs graves, qui devront généralement conduire à l'arrêt du programme
* java.lang.Exception pour des événements inattendus, qui seront souvent traités de sorte qu'elle ne provoque pas l'arrêt du programme

### Traitement des exceptions en Java

Pour traiter les erreurs, Java propose un mécanisme d'exception, consistant à effectuer les instructions dans un bloc d'essai (le bloc **try**) qui surveille les instructions. Lors de l'apparition d'une erreur, celle-ci est lancée dans un bloc de traitement d'erreur (le bloc **catch**, appelé **handler d'exception**) sous forme d'un objet appelé Exception.

Voici un exemple de gestion d'exception :

class MaClasse {  
  
 public static void main(String[] args) {  
  
 // Instructions inoffensives (affectations, ...);  
 try {  
 // Instructions susceptibles de provoquer des erreurs;  
 } catch (TypeException loE) {  
 // Instructions de traitement de l'erreur;  
 }  
 // Instructions si aucune erreur est apparue;  
  
 }   
}

### Checked et Unchecked Exception

Il est nécessaire de bien distinguer dans une application les exceptions checked et unchecked. Dans Java, une distinction est faite entre :

* les exceptions déclarées dans la signature d'une méthode (Checked Exception), par exemple void foo () throws ThisExceptionType { ... }
* les exceptions à l'exécution (RuntimeException ou Unchecked Exception), qui correspondent à des événements impossibles à localiser lexicalement à la compilation (les exceptions asynchrones), ou pouvant survenir à tout moment dans l'exécution du programme, comme les problèmes d'allocation mémoire

Les "checked exceptions" essayent de résoudre un problème de contrat. L'interface d'un module (d'une bibliothèque de classes) représente un contrat entre l'auteur du module et son utilisateur : l'argument est qu'un tel contrat ne devrait pas passer sous silence les exceptions susceptibles d'être propagées hors des frontières du module.

En spécifiant les exceptions dans les signatures des méthodes, on introduit toutefois un problème. En effet, les méthodes clientes doivent choisir dans l'alternative :

* installer un Gestionnaire d’Exceptions (GE) pour les exceptions du module, ou bien
* déclarer à leur tour ces exceptions.

Les méthodes utilisant des checked exceptions contaminent leurs clients avec l'obligation de décorer leur signature, s'ils n'installent pas de GE pour ces exceptions. Cette contamination trahit en partie le contrat d'indépendance entre le lieu du signalement d'une exception et le lieu de son traitement, en exposant toutes ces déclarations d'interfaces dans le chemin d'appel ; en somme elles nous ramènent aux inconvénients des langages de programmation sans GE (transmission de l'exception par une valeur de retour spéciale, prévue en tout point de la chaîne d'appels). Les "checked exceptions" violent finalement la philosophie des exceptions (*la non-localité entre le lieu de la condition et le lieu de son traitement*).

Ainsi, la bibliothèque standard de Java utilise en pratique des runtime exceptions pour les opérateurs les plus courants (arithmétique, collections, allocation mémoire) afin éviter la pollution lexicale des checked exception.

### Préconisations

#### Choix entre les checked et unchecked exceptions

Dans une application J2EE en couches, on préconise de n'utiliser **des checked exceptions que pour les exceptions fonctionnelles**. Ainsi, tous les méthodes des services métiers ou des businesse delegate susceptibles de lancer une exception fonctionnelle (c'est-à-dire susceptible de signaler la violation de règles métier) devront le déclarer dans leur signature :

public interface IService {  
  
 public Client creer(Client client) throws BusinessException;  
  
}

Ainsi toute classe appelant une méthode métier (typiquement la couche présentation) pourra savoir que la méthode est susceptible de lancer une exception fonctionnelle et pourra ainsi choisir de la gérer comme bon lui semble (typiquement avec un gestionnaire d'exception qui récupèrera le message de l'exception et l'affichera à l'utilisateur).

Dans la couche présentation (dans un backing bean par exemple), on aura donc un code comme celui-ci :

ISservice service = getServiceFactory().getService();  
try {  
 service.creerClient(client);  
} catch(BusinessException e) {  
 // code pour transformer l'exception en messages, redirection vers la page appropriée  
 saveBusinessException(e);  
}

Si on ne gère pas les exceptions fonctionnelles de façon spécifique, elles seront traitées par la couche présentation comme des erreurs techniques, ce qui risque de perturber l'utilisateur.

En revanche, pour tout ce qui concerne les exceptions techniques, on privilégiera les unchecked exceptions. Cela d'une part pour rester cohérent avec les frameworks techniques (Spring et Hibernate 3+ n'utilisent que des unchecked exceptions) et d'autre part parce que la plupart du temps, si une exception technique est levée dans un composant d'une couche donnée, celui-ci ne peut pas gérer l'exception. Si par exemple la connexion à la base de données échoue, les premiers composants concernés sont les DAOs de la couche persistance. Un DAO ne peut fonctionner sans connexion BD et n'est pas censé essayer d'en récupérer une. L'échec de connexion doit donc être remonté aux couches supérieures (la plupart du temps jusqu'à la couche présentation qui gère d'une façon générique les exceptions techniques, avec une journalisation et un message d'erreur à l'utilisateur.

#### Gestion de certaines exceptions

Les framework techniques et le développement en couches permettent au développeur de ne pas trop se soucier de la gestion des exceptions. Il arrive cependant que l'on soit obligé de gérer soit-même une exception technique. Par exemple, Hibernate propose un système de gestion de concurrence. Il lance une StaleStateException automatiquement. Ce type d'erreur est assez ambigüe : on ne peut pas la traiter comme une exception technique, il est préférable de signaler l'erreur à l'utilisateur, comme une exception fonctionnelle ou alors on peut choisir de traiter l'exception dans un service, en essayant de faire un merge (fusionner les 2 versions de l'objet qui n'est pas à jour). Si on choisit cette dernière option, on peut se retrouver avec le code suivant dans la coucher service :

public classe Service implements IService {  
  
 public Client update(Client client) {  
 try {  
 getClientDAO().update(client);  
 } catch(StaleStateException e) {  
 // on tente de faire un merge pour inclure les modifications  
 }  
 }  
}

Avec un tel code, on se retrouve avec une dépendance de la couche métier vers l'implémentation de la couche de persistance, chose qui est censée être évitée par le pattern de DAO. La solution consiste donc à intercepter toutes les StaleStateException et les relancer sous la forme d'une exception indépendante de l'implémentation (ex. : VersionException), et cela avant que les exceptions n'arrivent à la couche service.

Le framework d'AOP de Spring permet de faire cela en plaçant entre les couches des intercepteurs gérant les exceptions. Ce fonctionnement est explicité dans la partie sur Spring.

### Exceptions métier

Il existe une classe de base pour les exceptions métier : BusinessException. Le principe des exceptions est très utile puisqu'il permet d'interrompre l'exécution mais en revanche il ne permet de disposer que d'un objet : l'exception. Il est donc utile de pouvoir disposer de plusieurs erreurs métier dans une même exception, d'où l'existence d'une classe CompositeBusinessException, héritant de BusinessException. On peut donc facilement ajouter des erreurs à une telle exception, par exemple dans un Checker :

public verification(Client client) throws BusinessException {  
 CompositeBusinessException ex = new CompositeBusinessException();  
 // code métier  
 ...  
  
 if(pasValide) {  
 ex.addBusinessException("Cette règle métier n'est pas vérifiée !");  
 }  
  
 ...  
  
 if(toujoursPasValide) {  
 ex.addBusinessException("Encore une règle métier non-vérifiée !");  
 }  
  
 // des erreurs ?  
 if(!ex.isEmpty()) {  
 throw ex;  
 }  
}

**NB : pour des raisons de simplification, le code précédent n'est pas internationalisé (il est essentiel de ne pas coder en dur les messages d'erreur.** (cf. [Gestion des Messages](http://sunic.mediapost.fr/dsiwiki/index.php?n=Transverse.Messages))

## Exception Métier

A chaque exception métier correspond un libellé de message d'erreur.  
Dans le prototype, un fichier de ressources pour ce type de message a été créé (voir le fichier de configuration municipostPos-service.xml) :

<!-- le bundle des messages des exceptions métier-->  
 <bean id="messageSourceBusiness"  
 class="org.springframework.context.support.ResourceBundleMessageSource">  
 <property name="basenames">  
 <list>  
 <value>fr.mediapost.municipost.service.businessMessages</value>  
 </list>  
 </property>  
 </bean>

Le fichier de messages fr.mediapost.municipost.service.businessMessages.properties contient des paires clé/valeur du type :

msg.exception.0001=la liste des clients est vide

Chaque service métier se voit injecter le fichier de messages. (Il s'agit en fait du bean parent dont héritent tous les services)

<!-- le service abstrait -->  
 <bean id="templateService" abstract="true" class="fr.mediapost.municipost.service.MunicipostAbstractService">  
 <property name="bundle" ref="messageSourceBusiness" />   
 </bean>  
  
 <!-- les services -->  
 <bean id="serviceClient" class="fr.mediapost.municipost.service.impl.ServiceClientDefaultImpl" parent="templateService">  
 <property name="clientDAO" ref="rcCliDao" />   
 <property name="juridiqueDAO" ref="rcJuriDao" />  
 </bean>

Ainsi chaque service peut accéder à ce fichier de messages.  
**msg.exception.0001** est le code du libellé du message d'erreur.

public List findClientByNameAndPage(String aName, int page) throws BusinessException{  
  
 List listClients = clientDAO.findClientByNameAndPage(aName,page);  
 if ((listClients == null) || (listClients.isEmpty()))  
 {  
 // création des erreurs fonctionnelles  
 BusinessException ex = new BusinessException(getBundle().getMessage("msg.exception.0001",null,null));  
 throw ex;  
 }  
 return listClients;  
 }

## Gestion des transactions

voir également [la gestion des transactions avec Spring](http://sunic.mediapost.fr/dsiwiki/index.php?n=Transverse.Spring).

### Principes

Une transaction transforme une base de données (par extension tout type d’objet ou d’ensemble d’objets) d’un état consistant vers un autre état consistant. Les transactions se déroulent de façon isolée, protégées des autres transactions concurrentes. Une transaction assure que la totalité de la tâche soit effectuée de façon définitive (commit), sinon que rien ne soit fait (rollback, en cas d’erreur par exemple). Pour résumer, les transactions partagent toutes 4 principes, résumés par l'acronyme ACID :

* Atomicité : Une transaction est indivisible, elle est soit exécutée complètement soit pas du tout, mais pas à moitié.
* Cohérence : Une transaction assure l'intégrité des données, elle transfère des données d'un état cohérent à un autre. Si la procédure de mise à jour se passe mal, elle restaure les données initiales. La base reste cohérente même en cas de problèmes.
* Isolation : Cette propriété assure que des transactions concurrentes n'interférent pas entre elles. Si plusieurs transactions sont exécutées en même temps, elles se déroulent séquentiellement.
* Durabilité : Stipule que les résultats d'une transaction qui s'est effectuée correctement sont permanents et ne peuvent pas être affectés par une panne quelconque du système.

### Niveaux d’isolation

Le niveau d’isolation est obligatoire et important. Il est positionné sur la connexion comme paramètre général de la base de données mais aussi au niveau de la couche données métier et transactionnelle (couche logique métier). Le niveau d’isolation géré par la base de données est le minimum. Si ce niveau est laxiste (dirty read) pour les couches logicielles, il faudra prévoir de développer la gestion des exceptions de la base de données. Pour éviter cela mais aussi des accès inutiles à la base de données, le niveau d’isolation choisi pour la base de données est à reporter sur les couches logicielles. En regard du standard ANSI SQL92, il peut y avoir 3 types de problèmes relatifs aux transactions concurrentes et dans tous les cas les valeurs sont fausses du point de vue de la lecture :

* Lecture sale ou " dirty read " : Une transaction lit des données contenant un changement non validé d'une autre transaction. Une partie des données peut se révéler fausse en fonction du commit ou du rollback de l'autre transaction.
* Lecture non répétable ou " nonrepeatable reads " : Une transaction lit une donnée, une seconde transaction change la même donnée et la première transaction relit la donnée et obtient une valeur différente. La donnée a changé et est incohérente sur la transaction.
* Lecture fantôme ou " phantom reads " : Dans une transaction on ré-exécute une requête, retournant un ensemble de données qui satisfont une condition de recherche. Un nouveau jeu de données apparaît entre les deux opérations de lecture.

Les niveaux d'isolation suivants sont généralement supportés par les SGBDR ou les systèmes de données d'entreprise (EIS) et ainsi acceptés par les " ressource manager " :

* TRANSACTION\_READ\_UNCOMMITTED : La transaction peut lire des données non validées, c'est à dire des données qui ont été modifiées et non validées par des transactions concurrentes. Toutes les erreurs vues ci-dessus peuvent arriver.

Ce niveau est très dangereux dans les environnements stratégiques où des transactions simultanées mettent à jour des données partagées. Il est également inapproprié pour tous les calculs sensibles, comme les opérations de débit/crédit sur comptes bancaires qui doivent adopter un mode d'isolation plus strict.

Ce niveau d'isolation reste approprié si vous savez à l'avance qu'une instance de votre composant ne fonctionne qu'en l'absence de toute autre transaction simultanée. Cependant, dans la plupart des contextes transactionnels, ce niveau d'isolation est insuffisant.

Son principal avantage est la performance, car le système transactionnel sous-jacent n'a pas besoin de verrouiller les données partagées.

* TRANSACTION\_READ\_COMMITTED : Ce niveau d'isolation fait que l'on ne peut pas lire les changements non validés des transactions concurrentes.

Ce niveau est certainement plus robuste que le mode précédent. Vous ne pouvez plus lire les données écrites, et non validées, ce qui signifie par conséquent que toutes les données que vous lisez sont cohérentes.

Ce mode est fréquemment utilisé pour les programmes qui effectuent des lectures en base pour constituer des rapports sur les valeurs des données.

* TRANSACTION\_REPEATABLE\_READ : Ce niveau d'isolation assure en plus que lire les données plusieurs fois retournera les mêmes valeurs même si une autre transaction modifie ces données. Les lectures sont répétables.

Ce mode est utile lorsque l'on doit mettre à jour un ou plusieurs éléments de données d'une ressource, comme un ou plusieurs enregistrements d'une BDDR. Il est nécessaire de lire toutes les lignes et de les mettre à jour, en sachant qu'aucune n'est modifiée par d'autres transactions simultanées. Si l'on choisit de relire l'une de ces lignes ultérieurement au cours de la transaction, on a la certitude qu'elle contient les mêmes données qu'au début de la transaction.

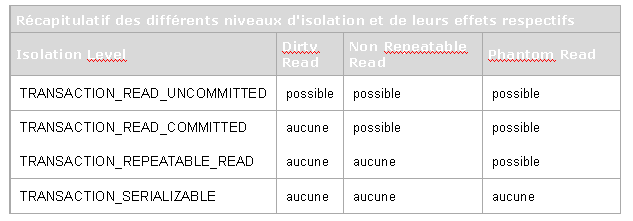
* TRANSACTION\_SERIALIZABLE : La transaction a les privilèges exclusifs en lecture/ecriture sur des données en les bloquant; les autres transactions ne peuvent ni lire ni écrire sur ces données. C'est le niveau d'isolation le plus contraignant empêchant également les " phantom reads ".

Ce mode doit être utilisé pour les systèmes stratégiques qui exigent une parfaite isolation transactionnelle. Il assure que les données lues ont été validées, que la lecture reste identique au fil du temps et que de mystérieuses données validées n'apparaissent pas dans la base en cours de traitement en raison de transactions simultanées.

Ce niveau d'isolation doit être utilisé avec parcimonie, car il induit un coût certain. Si toutes les opérations sont réalisées dans ce mode, les performances de la base de données ont tendance à se dégrader très rapidement jusqu'à l'immobilisation éventuelle.

Les erreurs transactionnelles relatives aux 3 problèmes cités de lecture erronée pouvant s'avérer très difficiles à détecter, il est également bon d'utiliser ce mode pour s'en détacher complètement.

Récapitulatif des différents niveaux d'isolation et de leurs effets respectifs:



En dehors de cas de figure exceptionnels dûment commentés Mediapost préconise d'utiliser le niveau READ\_COMMITED. Il s'agit du niveau par défaut du moteur Sybase

### Implémentation

Il existe différentes stratégies de gestion des exceptions en environnement J2EE :

* Gestion déclarative des transactions : la délimitation des frontières des transactions (début / commit / rollback) ne sont pas fait de manière « programmative », mais par le positionnement d’attributs transactionnels pour les méthodes de composants. C’est le conteneur de composant qui gère ensuite le cycle de vie des transactions. Cela est mis en œuvre dans les conteneurs EJBs, mais également dans certains conteneurs légers, type Spring.
* Gestion programmative des transactions : la délimitation des frontières des transactions (début / commit / rollback) est faite « en dur » dans le code des composants. On va distinguer à encore 2 types de transactions :
  + Transactions JDBC : la transaction est gérée par le Transaction Manager du SGBD. La délimitation des frontières des transactions se fait sur l’objet java.sql.Connection. Pour pouvoir effectuer des transactions distribuées, le driver JDBC doit proposer une implémentation de javax.sql.XAConnection, et le Transaction Manager du SGBD doit pouvoir les supporter.
  + Transactions JTA : la transaction est gérée par le Transaction Manager du serveur d’applications J2EE. La délimitation des frontières des transactions se fait via un objet javax.transaction.UserTransaction, récupérable via JNDI ou via l’EJBContext.

Quel que soit le type de gestion des transactions retenu (dépendant du serveur d’application, des technologies mises en œuvre, de la nécessité de transactions distribuées), la cohérence des traitements transactionnels constitue un point clé du projet. Dans une transaction, un ensemble de composants et de traitements peuvent concourir.

Dans le cas d’une gestion des transactions déclaratives, différents attributs transactionnels sont disponibles permettant de répondre aux problématiques suivantes. Les différents noms de ces attributs dépendent encore souvent du serveur d’application (dans notre cas, le vocabulaire utilisé provient de la norme EJB).

* Attribut Required :

Le bean nécessite une transaction. Si elle existe, il s’intègre à celle en cours sinon il en crée une nouvelle. Permet une grande flexibilité, et en fonction du scénario de rejoindre ou non une transaction existante.

* Attribut Supports :

Le bean s’intègre à la transaction en cours si elle existe sinon aucune transaction n’est créée. Le bean s’exécute alors sans transaction. A utiliser avec précaution car il faut éviter d'utiliser le bean en dehors de toute transaction. A éviter pour les opérations critiques.

* Attribut Mandatory :

Une transaction doit être en cours sinon l’exception javax.ejb.TransactionRequired est émise. Cette méthode force le client à déclarer lui même une nouvelle transaction. Garantit que le bean ne peut être appelé que dans le cadre d'une transaction, cependant nécessite un tiers pour initier la transaction avant l'appel du bean. Utile lorsqu'il est appelé par d'autres beans ou lorsque le bean fonctionne au sein d'un système plus large, dans lequel il fait partie d'un ensemble important d'opérations.

* Attribut Not supported :

Le bean ne peut utiliser aucune transaction ou impacter une transaction. Il est dangereux de l’utiliser au sein d’une transaction afin de préserver les propriétés ACID (Atomicité Cohérence Isolation Durabilité). A ne pas utiliser avec des EJB entités ! Doit être employé que si le bean n'as pas besoin de propriétés transactionnelles ACID. Soit lorsque le bean ne réalise pas d'opérations critiques et si son isolation vis-à-vis d'autres opérations simultanées n'est pas une nécessité.

* Attribut Never :

La méthode sera invoquée sans contexte transactionnel que s'il n'existe pas de contexte transactionnel parent sinon une erreur est lancée. Pour s'assurer que les clients utilisant le bean n'utilisent pas de transaction et réduire ainsi les erreurs.

* Attribut Requires New :

Le bean crée une nouvelle transaction à chaque appel. Il ne peut s’intégrer à une transaction en cours. Ainsi, les transactions sont isolées contrairement aux autres attributs. Utile si le bean doit bénéficier de propriétés transactionnelles ACID, tout en fonctionnant en tant qu'unité de travail unique, sans que des logiques externes puissent interférer dans la même transaction.

## Utilisation d'ACEGI

Une présentation de l'outil ACEGI est disponible dans le wiki : [Présentation ACEGI](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.ACEGI)  
Les diverses solutions de gestion de l'identification et des autorisations est disponible dans le wiki : [Gestion de la sécurité](http://sunic.mediapost.fr/dsiwiki/index.php?n=Jsf.Securite)  
La description de la configuration ACEGI mise en place dans ce contexte est disponible dans le wiki : [Configuration ACEGI](http://sunic.mediapost.fr/dsiwiki/index.php?n=Securite.Configuration)

Les points abordés sont les suivants:  
[Configuration](#scroll-bookmark-352)  
[Sources de données d’identification et d’autorisation](#scroll-bookmark-353)  
[L’annuaire pour gérer les utilisateurs](#scroll-bookmark-354)  
[Processus d’identification](#scroll-bookmark-355)  
[Ajouter un droit applicatif](#scroll-bookmark-356)  
[Ajouter un groupe](#scroll-bookmark-357)  
[Ajouter un utilisateur](#scroll-bookmark-358)  
[Affecter un groupe à un utilisateur](#scroll-bookmark-359)  
[Vérifier les droits en Java](#scroll-bookmark-360)  
[Vérifier les droits en JSF](#scroll-bookmark-361)  
[Restreindre l’accès à une vue](#scroll-bookmark-362)  
[Restreindre l’accès à une méthode](#scroll-bookmark-363)  
[Restreindre le canal d’accès à une vue](#scroll-bookmark-364)  
[Test unitaires](#scroll-bookmark-365)  
[Roadmap d'ACEGI](#scroll-bookmark-366)  
[Spring LDAP](#scroll-bookmark-367)

### Configuration

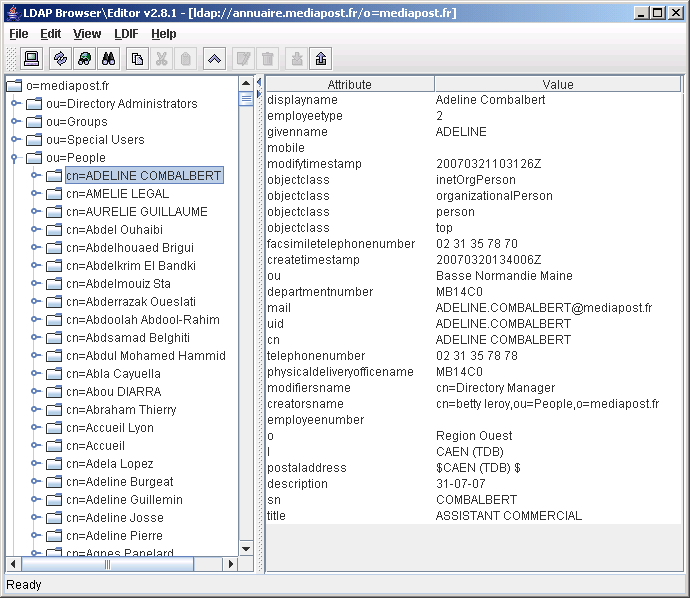
La mise en place d’ACEGI dans l’architecture a fait l’objet d’un prototype disponible dans CVS : Municipost\_facelets.  
Un archétype maven permet également de commencer un projet utilisant ACEGI ; il est également dans CVS. [Les archétypes](http://sunic.mediapost.fr/dsiwiki/index.php?n=Maven.ArchetypesMediapost)  
La description de la configuration ACEGI mise en place dans ce contexte est disponible dans le wiki : [Configuration ACEGI](http://sunic.mediapost.fr/dsiwiki/index.php?n=Securite.Configuration)

### Sources de données d’identification et d’autorisation

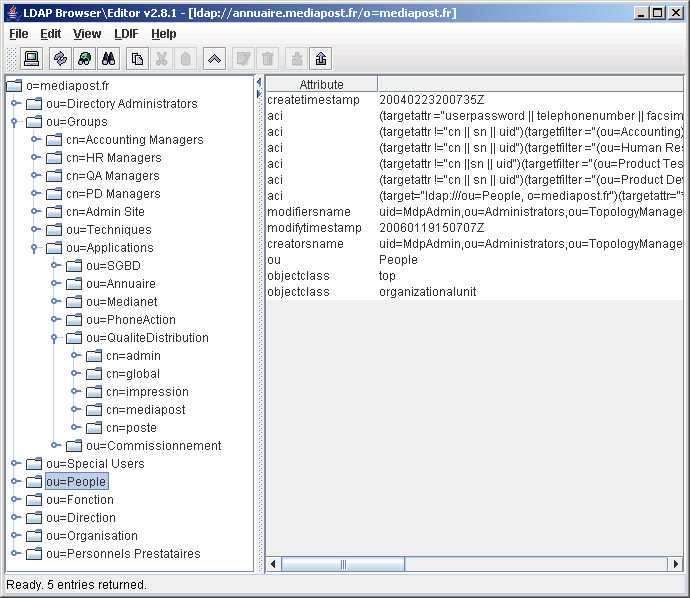
Pour la gestion des identités et des autorisations dans les applications JEE Mediapost a fait le choix de centraliser le plus d’informations dans l’annuaire LDAP d’entreprise (ldap://annuaire.mediapost.fr/o=mediapost.fr).

#### L’annuaire pour gérer les utilisateurs

Cet annuaire définit les utilisateurs dans la branche ou=People, o=mediapost.fr. C’est grâce à ces informations que les utilisateurs sont authentifiés. Ces utilisateurs sont affectés (de façon dynamique ou statique) à des groupes pour chaque application.

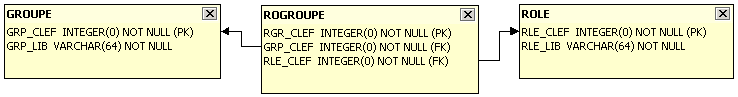


Les groupes applicatifs sont définis dans la branche ou=Applications, ou=Groups, o=mediapost.fr. Chaque application a une entrée de type ou dans cette branche. Cette entrée contiendra les groupes propres à cette application. Le nom du groupe est identifié grâce à l’attribut cn du groupe.



#### Une base de données pour gérer les droits applicatifs

Chaque application va devoir définir un certain nombre de droits applicatifs (souvent appelés rôles). Ce sont ces éléments qui permettent de tester dans l’application si une fonctionnalité / un affichage est disponible pour l’utilisateur. Chaque application a des besoins spécifiques en fonction de la finesse de gestion voulue. C’est pourquoi il a été choisi de stoker ces droits applicatifs dans une base de données propre à l’application. Ainsi, tout nom de groupe doit être associé à une collection de droits applicatifs dans cette base de données.

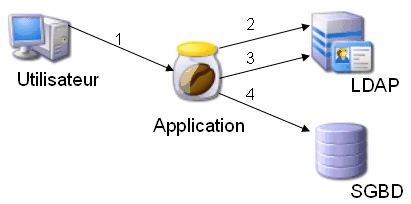


(Cette structure est celle mise en place dans l’archétype)

### Processus d’identification

L’identification des utilisateurs commence par le login à l’aide d’un formulaire ou de NTLM (1). Plus tard, une architecture SSO sera mise en place. A partir des données de ce formulaire ACEGI lance l’identification via l’annuaire (2). Si le résultat est positif, ACEGI lance l’outil de chargement des droits applicatif présent dans le framework. Ceci se déroule en étapes :

* Recherche des groupes affectés à l’utilisateur dans l’annuaire (3)
* Chargement des rôles à partir des noms de groupe dans la base de données (4)



1. Saisie du formulaire ou négociation NTLM (selon la configuration)
2. Identification LDAP
3. Recherche des groupes de l’utilisateur
4. Recherche des rôles associés aux groupes

### Ajouter un droit applicatif

Comme vu précédemment, les droits applicatifs sont définis dans la base de données. Pour en ajouter un il faut :

* Ajouter une entrée dans la table de définition des droits applicatifs (table ROLE de l’exemple)
* Ajouter les associations nécessaires avec les groupes donnant accès à ce droit (table ROGROUPE de l’exemple)

### Ajouter un groupe

Comme vu précédemment, les groupes sont définis à deux endroits :

* dans l’annuaire pour les associer aux utilisateurs
* dans la base de données pour les associer aux droits applicatifs

Il est nécessaire de faire l’ajout dans ces deux sources de données pour que le nouveau groupe soit pris en compte.

Pour modifier l’annuaire il est possible d’ajouter des éléments par import d’un fichier au format LDIF. Pour plus d’informations sur ce format vous pouvez consulter les liens suivants : [Wikipedia LDIF](http://fr.wikipedia.org/wiki/Ldif) [RFC2849](http://tools.ietf.org/html/rfc2849)

Exemple :

#---------------  
# création de la racine des groupes pour l'application  
#---------------  
dn: ou=MunicipostPos, ou=Applications, ou=Groups, o=mediapost.fr  
ou: MunicipostPos  
objectclass: top  
objectclass: organizationalunit  
description: Prototype utilisant l'identification LDAP  
  
  
#---------------  
# Groupe admin  
#---------------  
dn: cn=admin, ou=MunicipostPos, ou=Applications, ou=Groups, o=mediapost.fr  
objectclass: top  
objectclass: groupofuniquenames  
objectclass: groupofurls  
cn: admin  
description: Utilisateurs ayant la possibilité d'administrer  
 les informations du site  
  
  
#---------------  
# Groupe utilisateurs  
#---------------  
dn: cn=usr, ou=MunicipostPos, ou=Applications, ou=Groups, o=mediapost.fr  
objectclass: top  
objectclass: groupofuniquenames  
objectclass: groupofurls  
cn: usr  
description: Utilisateurs pouvant accéder au site  
  
#fin de fichier

[Outil de modification de l’annuaire](http://sunic.mediapost.fr/dsiwiki/index.php?n=Securite.EditLdap)

### Ajouter un utilisateur

L’ajout d’un utilisateur dans l’annuaire doit être fait par le service en charge de l’administration.

### Affecter un groupe à un utilisateur

Pour ajouter un membre à un groupe existant il faut modifier l’annuaire. Comme pour la création d’un groupe ceci peut être fait à l’aide d’un fichier LDIF Les groupes définis peuvent contenir soit des membres définis de façon statique soit des membres définis de façon dynamique.

#### Ajout d’un membre de façon statique

Pour cela il faut ajouter une valeur à l’attribut uniquemember. Cette valeur doit être le contenu de l’attribut dn (équivalent de la clef primaire pour une base de données) de l’utilisateur.

Exemple :

#---------------  
# Groupe admin  
#---------------  
dn: cn=admin, ou=MunicipostPos, ou=Applications, ou=Groups, o=mediapost.fr  
changetype: modify  
add: uniquemember  
uniquemember: cn= Admin MunicipostPos, ou=People, o=mediapost.fr  
#fin de fichier

[Outil de modification de l’annuaire](http://sunic.mediapost.fr/dsiwiki/index.php?n=Securite.EditLdap)

#### Ajout d’un membre de façon dynamique

L’ensemble des membres définis dynamique est en fait l’équivalent du résultat d’un filtre de recherche LDAP. Comme pour la façon statique il faut ajouter une valeur à un attribut multivalué ; l’attribut memberurl.

Exemple :

#---------------  
# Groupe utilisateurs  
#---------------  
dn: cn=usr, ou=MunicipostPos, ou=Applications, ou=Groups, o=mediapost.fr  
changetype: modify  
add: memberurl  
memberurl: ldap:///cn=dyn4 MunicipostPos, ou=People, o=mediapost.fr??base?objectclass=person  
  
#fin de fichier

[Outil de modification de l’annuaire](http://sunic.mediapost.fr/dsiwiki/index.php?n=Securite.EditLdap)

### Vérifier les droits en Java

Dans le cadre de l’exécution au sein du cycle de vie JSF il est possible d’accéder au contexte de sécurité d’ACEGI. En effet, un Listener initialise le contexte de sécurité à partir de la session utilisateur, le rendant accessible à tout traitement du Thread. A partir de ce contexte nous avons directement accès aux données d’identification (Authentication) puis d’autorisation.

Exemple :

SecurityContext acegiCtxt = SecurityContextHolder.getContext();  
 Authentication ident = acegiCtxt.getAuthentication();GrantedAuthority[] autor = ident.getAuthorities();

A noter, les fonctions EL présentées ci-dessus sont également accessible de façon statique.

Exemple :

fr.mediapost.fwk.security.acegi.util.TagLibFunctions.isUserInAllRoles("r\_admin,r\_client");

### Vérifier les droits en JSF

Afin de pouvoir tester les droits de l’utilisateur au sein du code des vues, un ensemble de fonctions EL ont été mise en place. Ces fonctions renvoient un booléen :

* isUserInAllRoles renvoie vrai si l’utilisateur a tous les rôles
* isUserInAnyRoles renvoie vrai si l’utilisateur a au moins un des rôles
* isUserInNoRoles renvoie vrai si l’utilisateur n’a aucun des rôles

L’argument passé à ces fonctions est une chaîne de caractère contenant des rôles séparés par une virgule.

Pour avoir accès à ces fonctions vous devez utiliser le taglib de namespace http://sourceforge.net/projects/jsf-comp/acegijsf

Exemple :

…xmlns:authz=http://sourceforge.net/projects/jsf-comp/acegijsf …  
 <h:outputText value="#{messages['page2.success']}" rendered="#{authz:isUserInAllRoles('r\_usr,r\_admin')}" />  
 <h:outputText value="#{messages['page2.echec']}" rendered="#{authz:isUserInNoRoles('r\_usr,r\_admin')}" />

### Restreindre l’accès à une vue

ACEGI utilise un filtre pour restreindre l’accès aux vues. En plus de ce filtre le ViewHandler mis en place fait également appel à ce filtre à la création de vues. Pour ajouter une restriction vous devez modifier votre version du fichier de configuration Spring relatif à ACEGI. Attention, les restrictions sont évaluées dans l’ordre de leur déclaration. La restriction retenue sera la première déclaration correspondant à la vue demandée.

Exemple :

…  
 <bean id="filterInvocationInterceptor"  
 …  
 /faces/\*client.jsf\*=r\_client,r\_admin

Pour plus de détail voir [Configuration ACEGI](http://sunic.mediapost.fr/dsiwiki/index.php?n=Securite.Configuration)

### Restreindre le canal d’accès à une vue

ACEGI permet de restreindre le canal (http ou HTTPS) accédant à une vue. Pour ajouter une restriction vous devez modifier votre version du fichier de configuration Spring relatif à ACEGI. Attention, les restrictions sont évaluées dans l’ordre de leur déclaration. La restriction retenue sera la première déclaration correspondant à la vue demandée.

Exemple :

…  
 < <bean id="channelProcessingFilter"  
 …  
 \A/j\_acegi\_security\_check.\*\Z=REQUIRES\_SECURE\_CHANNEL

Pour plus de détail voir [Configuration ACEGI](http://sunic.mediapost.fr/dsiwiki/index.php?n=Securite.Configuration)

### Restreindre l’accès à une méthode

La mise en place d’ACEGI permet de restreindre l’accès aux méthodes des objets instanciés via Spring. Selon le choix de définition des restrictions d’accès aux méthodes vous pouvez soit :

* Modifier votre version du fichier de configuration Spring relatif à ACEGI
* Ajouter une annotation @Secured à la méthode concernée.

Exemple de déclaration dans la configuration :

…  
 <bean id="aspectJSecurityInterceptor"  
 …  
 fr.mediapost.municipost.integration.impl.HibernateClientDAO.save\*=r\_client

Exemple d’annotation :

@Secured({"r\_admin", "r\_client"})  
 public String getUserAge(){  
 …

Pour plus de détail voir [Configuration ACEGI](http://sunic.mediapost.fr/dsiwiki/index.php?n=Securite.Configuration)

### Test unitaires

Il est souhaitable que les tests unitaire utilisent la même configuration que l’application (sécurité y compris). De plus, il peut être nécessaire d’implémenter des tests unitaires dont le seul but est de vérifier la configuration de sécurité. Lors de l’exécution d’un test unitaire, le contexte de sécurité ACEGI doit être initialisé. Une classe d’outil est disponible dans ce but : AcegiTestIntegration. Les étapes à suivre sont :

1. Importer de façon statique les outils
2. Positionner le gestionnaire d’identification de test
3. Définir les droits applicatifs voulu

Un exemple est disponible dans CVS (projet Municipost\_facelets classe ServiceClientDefaultSecurityTest) et dans l’archétype dédié à ACEGI.

Exemple

…  
 (1)  
 import static fr.mediapost.fwk.security.acegi.util.AcegiTestIntegration.setAuthenticationToken.\*;  
 …  
 (2)  
 @Override  
 protected void setUp() throws Exception {  
 super.setUp();  
 // se positionner dans le mode de sécurité des testssetTestingAuthenticationProvider(this.getApplicationContext());  
 }  
 …  
 (3)setAuthenticationToken(new String[] { "role" });

### Roadmap d'ACEGI

Dans une prochaine version, un namespace Spring dédié à la sécurité va être proposé L’intégration d’ACEGI dans Spring n’est à ce jour (26/11/07) pas finalisée. Le namespace représente un espoir de simplification de la configuration.

### Spring LDAP

Il faut définir une configuration spring:

<bean id="contextSource" class="org.springframework.ldap.core.support.LdapContextSource">  
 <property name="url" value="ldap://webservdev:389" />  
 <property name="base" value="o=mediapost.fr" />  
 </bean>  
  
 <bean id="ldapTemplate" class="org.springframework.ldap.core.LdapTemplate">  
 <constructor-arg ref="contextSource" />  
 </bean>  
  
 <bean id="utilisateurDaoLdap" class="fr.mediapost.commedia.metier.habilitation.integration.ldap.LDAPUtilisateurDAO">  
 <property name="ldapTemplate" ref="ldapTemplate" />  
 </bean>

Il faut ensuite créer un DAO:

Par exemple:

public class LDAPUtilisateurDAO implements IUtilisateurDAO{

private LdapTemplate ldapTemplate;  
 public List<Utilisateur> findUtilisateursByGroupe(Groupe groupe) {AndFilter filter = new AndFilter();  
 filter.and(new EqualsFilter("objectclass", "person"));  
 filter.and(new EqualsFilter("memberurl", "url: ldap://webservdev.mediapost.fr:389/cn=Administrateur, ou=Commedia,   
 ou=Applications,ou=Groups, o=mediapost.fr  
 memberurl: ldap:///o=mediapost.fr??sub?(&(objectclass=person)(title=\*"+groupe.nom+"\*))"));  
 return ldapTemplate.search(  
 "", filter.encode(),  
 new UtilisateurAttributesMapper() {  
 public Object mapFromAttributes(Attributes attrs)  
 throws NamingException {  
 return attrs.get("cn").get();  
 }  
 });  
 }

Il est également possible de créer un AttributeMapper personnalisé:

private class UtilisateurAttributesMapper implements AttributesMapper {  
 public Utilisateur mapFromAttributes(Attributes attrs) throws NamingException {  
 Utilisateur person = new Utilisateur();  
 person.setFullName((String)attrs.get("cn").get());  
 person.setLastName((String)attrs.get("sn").get());  
 person.setDescription((String)attrs.get("description").get());  
 return person;  
 }  
 }

## Spring

La version de Spring utilisée est la 2. Spring est utilisé sur les points suivants :

* conteneur de bean (définition de la partie métier dans un fichier XML)
* gestion déclarative des transactions
* utilisation de l'AOP pour relancer les exceptions
* accès aux données
  + utilisation des classes de base HibernateDaoSupport et JdbcDaoSupport
  + utilisation de l'API JDBC (JdbcTemplate) pour les instructions SQL

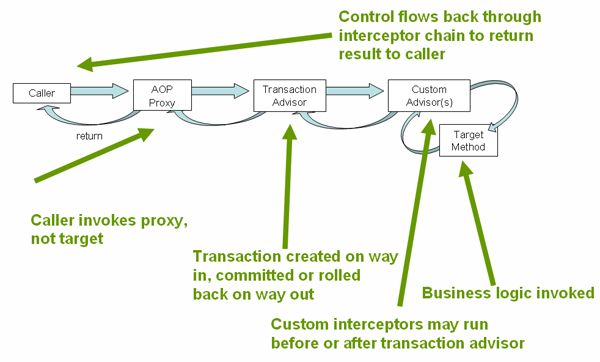
### Gestion des transactions avec Spring

Voir [quelques explications d'ordre général sur les transactions](http://sunic.mediapost.fr/dsiwiki/index.php?n=Transverse.Transaction).

Spring permet de gérer les transactions de façon *déclarative*, c'est-à-dire qu'aucune instruction de gestion de transaction n'apparaît dans le code. Cela présente les avantages suivants par rapport à une approche programmatique :

* le code est plus clair et plus fiable
* les objets transactionnels (dans notre cas les services) sont des POJOs (ils n'ont pas de dépendances vis-à-vis d'une API de gestion de transaction)
* la politique transactionnelle n'est pas figée dans le code, les services sont donc réutilisables
* on peut changer d'implémentation de gestion de transaction sans changer de code. La gestion transactionnelle peut ainsi être gérée par Spring ou par le serveur d'applications s'il en est capable.

C'est l'approche déclarative qui a été choisie pour le framework. La gestion des transactions se fait au niveau des services métier, ce qui permet d'englober dans une même transaction de nombreux appels à des DAO. Techniquement, la gestion des transactions se fait avec le framework de programmation orientée aspect (AOP) de Spring. Cela signifie que les objets transactionnels (les services) ne le sont pas nativement : dans le code de ceux-ci, il n'existe aucune instruction de gestion de transaction. Les services sont rendus transactionnels avec un système de proxy : Spring va substituer aux instances de services des objets dénommés des proxies, implémentant les mêmes interfaces, qui vont intercepter tous les appels aux services et gérer les transactions selon la configuration faite dans le contexte applicatif. Voici un schéma (issu de la documentation Spring) expliquant ce mécanisme :



La gestion déclarative des transactions nécessite les acteurs suivants :

* un gestionnaire de transaction (transaction manager), dont l'implémentation change selon l'application (ex. : HibernateTransactionManager pour une application mono-datasource à base d'Hibernate).
* une définition des attributs transactionnels, c'est-à-dire sur qui doit-on appliquer des transactions et de quelles manières

L'interface du gestionnaire de transaction est propre à Spring. Toutes les implémentations fournies par Spring implémentent cette interface et sont interchangeables, selon les besoins.

Voici un exemple de configuration :

<!-- un service métier -->  
<bean id="serviceClient" class="fr.mediapost.municipost.service.impl.ServiceClientDefaultImpl">  
 <!-- injection de DAOs -->  
 ...  
</bean>  
  
<!-- gestionnaire de transaction Hibernate -->  
<bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">  
 <!-- session factory, initialisée autre part dans le contexte -->  
 <property name="sessionFactory" ref="sessionFactory"/>  
</bean>   
  
<!-- attributs transactionnels : -->  
<!-- définition de la politique de transaction selon les noms des méthodes des objets transactionnels -->  
<!-- passe par la définition d'un advice spécifique au framework de transaction -->  
<tx:advice id="txAdvice" transaction-manager="transactionManager">  
 <tx:attributes>  
 <!-- les méthodes suivantes nécessitent une transaction -->  
 <tx:method name="create\*" propagation="REQUIRED" />  
 <tx:method name="update\*" propagation="REQUIRED" />  
 <tx:method name="delete\*" propagation="REQUIRED" />  
 <!--   
 toutes les autre méthodes s'insérent dans le contexte transactionnel  
 et sont supposées en lecture seule, pour améliorer les performances  
 -->  
 <tx:method name="\*" propagation="SUPPORTS" read-only="true"/>   
 </tx:attributes>  
</tx:advice>  
  
<!-- définition des proxies -->  
<aop:config>   
 <!-- définition des méthodes de service -->  
 <aop:pointcut id="serviceMethodes" expression="execution(\* fr.mediapost.municipost.service..\*Service\*Impl\*.\*(..))"/>  
 <!-- transactions -->  
 <!-- utilisation de la définition des méthodes de services et de l'advice précédemment défini -->  
 <aop:advisor advice-ref="txAdvice" pointcut-ref="serviceMethodes"/>  
</aop:config>

Dans cet exemple de code, on définit d'abord un service métier qui est un simple POJO. C'est lui qui doit être rendu transactionnel. On peut imaginer dans une application à grande échelle la définition de plusieurs dizaines de services métier.

Vient ensuite la déclaration du gestionnaire de transaction, puis celle de l'advice transactionnel (tx:advice). Celui-ci définit les attributs transactionnels des objets qui vont être rendus transactionnels. On se base sur le nom des méthodes des objets transactionnels. Ainsi, les méthodes commençant par create, update, delete des services métier nécessitent obligatoirement une transaction (attribut REQUIRED) et toutes les autres méthodes (pattern \*) s'insèrent dans le contexte transactionnel (SUPPORTS).

En dernier lieu vient la configuration AOP du contexte Spring (aop:config). C'est à partir de cette configuration que Spring va créer des proxies et les greffer sur les bean appropriés. Avec la ligne

<aop:pointcut id="serviceMethodes" expression="execution(\* fr.mediapost.municipost.service..\*Service\*Impl\*.\*(..))"/>

on définit en fait l'ensemble des services métier. Avec la ligne

<aop:advisor advice-ref="txAdvice" pointcut-ref="serviceMethodes"/>

on dit à Spring d'appliquer l'action (*advice*) txAdvice sur les objets correspondant au pattern (*pointcut*) serviceMethodes. Le bean définissant le service métier

<bean id="serviceClient" class="fr.mediapost.municipost.service.impl.ServiceClientDefaultImpl">

correspondant au pattern fr.mediapost.municipost.service..\*Service\*Impl\*.\*(..)), un proxy gérant le comportement transactionnel va lui être substitué.

L'ensemble des acteurs (gestionnaire de transaction, instructions d'AOP) sont relativement fixes dans cette définition. Il faut principalement prendre garde à la définition des méthodes de services, qui est susceptible de changer d'une application à l'autre :

<aop:pointcut id="serviceMethodes" expression="execution(\* fr.mediapost.municipost.service..\*Service\*Impl\*.\*(..))"/>

et aux attributs transactionnels :

<tx:attributes>  
 <!-- les méthodes suivantes nécessitent une transaction -->  
 <tx:method name="create\*" propagation="REQUIRED" />  
 <tx:method name="update\*" propagation="REQUIRED" />  
 <tx:method name="delete\*" propagation="REQUIRED" />  
 <!--   
 toutes les autre méthodes s'insérent dans le contexte transactionnel  
 et sont supposées en lecture seule, pour améliorer les performances  
 -->  
 <tx:method name="\*" propagation="SUPPORTS" read-only="true"/>   
</tx:attributes>

qui soit peuvent être étoffés au fur et à mesure des développements, soit au contraire ne sont jamais modifiés et imposent une convention de nommage pour les méthodes des services métier.

#### Transactions distribuées

Dès lors que plusieurs datasources sont accédées en lecture/écriture au sein d'un même service, il est nécessaire de mettre en place une architecture supportant les transactions distribuées. Il faut alors se tourner vers l'API JTA, c'est-à-dire utiliser le gestionnaire de transaction JTA de Spring. Si le serveur d'application cible ne propose pas un support pour la gestion des transactions distribuées (Tomcat n'est par exemple qu'un conteneur Web, avec un ensemble de services limités), il est nécessaire d'embarquer un tel système dans l'application.

La gestion des transactions distribuées en Java passe par l'utilisation de l'API JTA (et donc principalement de la classe UserTransaction) mais aussi par l'utilisation de ressources implémentant la norme XA. Par *ressources*, on entend principalement des bases de données (mais on peut aussi y inclure les files JMS et tout connecteur JCA) et par *implémentant la norme XA*, on entend que le driver doit fournir les classes de la norme XA (notamment XAConnection, XADataSource et surtout XAResource). L'idée de la norme XA est de pouvoir disposer de transactions ACID (Atomicité Cohérence Isolation Durabilité) sur plusieurs ressources. Pour résumer, on peut dire que la norme XA a deux composantes : d'une part les ressources XA peuvent dialoguer avec le gestionnaire de transaction pour savoir quand effectuer un commit ou un rollback et d'autre part, le gestionnaire de transactions est capable d'effectuer une restauration suite à un crash. La restauration est possible grâce à la journalisation des transactions imposée par la norme XA.

Les spécifications JTA et XA sont complexes et leur connaissance n'est pas essentielle à un développeur. Néanmoins, il est intéressant d'avoir quelques notions sur ces mécanismes. Pour aller plus loin, consulter [l'article de Mike Spille](http://jroller.com/page/pyrasun?catname=%2FXA).

Voici un rapide survol des solutions pour gérer des transactions XA :

* [JOTM](http://jotm.objectweb.org/) : c'est un gestionnaire embarqué, notamment utilisé par le serveur d'applications Jonas. Il ne propose pas de système de journalisation des transactions et ne peut pas donc pas récupérer correctement les transactions en cours après un crash. Il contient aussi un certain nombre de bug, même s'il est censé être en version stable.
* [Jencks](http://jencks.org) : conteneur JCA léger, Jencks est en fait basé sur le gestionnaire de transaction de Geronimo. Se basant sur JCA, il est censé pouvoir gérer tout type de ressources XA (JDBC, JMS...)
* [SimpleJTA](http://www.simplejta.org) : gestionnaire de transaction JTA autonome. Son seul inconvénient vient de son système de journalisation, qui utilise une base de données (il faut donc une base de données supplémentaire, Oracle ou Derby si on souhaite utiliser SimpleJTA)
* [BTM](http://www.bitronix.be/Btm/) (pour Bitronix Transaction Manager) : projet assez récent (en version alpha en septembre 2006), Bitronix est un gestionnaire de transaction JTA. Son développement se focalise pour l'instant sur les ressources XA en JDBC (pas de support pour JMS ou JCA). Contrairement à Jencks et JOTM, qui proposent des wrappers XA, BTM ne fonctionne qu'avec des bases de données proposant le support XA. Il est certainement actuellement la meilleure solution pour la gestion de transactions XA 100% JDBC.

Tous ces systèmes s'intègrent à Spring.

Le support des transactions distribuées étant désactivé sur le serveur Sybase de Mediapost, BTM ne peut être utilisé. L'explication portera donc sur Jencks.

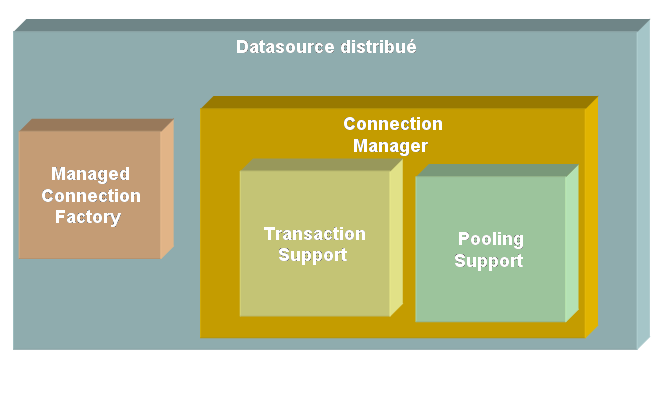
Voici la déclaration du gestionnaire de transaction :

<!-- creation du contexte JTA -->  
<bean id="transactionContextManager" class="org.jencks.factory.TransactionContextManagerFactoryBean"/>  
<bean id="userTransaction" class="org.jencks.factory.GeronimoTransactionManagerFactoryBean"/>  
<!-- gestionnaire de transactions -->  
<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">  
 <property name="userTransaction" ref="userTransaction"/>  
</bean>

La déclaration de la politique de gestion de transaction reste classique :

<!-- politique de transaction -->   
<tx:advice id="txAdvice" transaction-manager="transactionManager">  
 <tx:attributes>  
 <tx:method name="create\*" propagation="REQUIRED" />  
 <tx:method name="update\*" propagation="REQUIRED" />  
 <tx:method name="delete\*" propagation="REQUIRED" />  
 <tx:method name="\*" propagation="SUPPORTS" read-only="true"/>  
 </tx:attributes>  
</tx:advice>  
  
<aop:config>   
 <aop:pointcut id="serviceMethodes" expression="execution(\* fr.mediapost.municipost.spring..\*Service\*.\*(..))"/>  
 <aop:advisor advice-ref="txAdvice" pointcut-ref="serviceMethodes"/>  
</aop:config>

La définition d'un datasource est en revanche plus complexe :



<bean id="transactionSupport" class="org.jencks.factory.XATransactionFactoryBean">  
 <property name="useTransactionCaching" value="true" />  
 <property name="useThreadCaching" value="false" />  
</bean>  
  
<bean id="poolingSupport" class="org.jencks.factory.SinglePoolFactoryBean">  
 <property name="maxSize" value="2" />  
 <property name="minSize" value="1" />  
 <property name="blockingTimeoutMilliseconds" value="60" />  
 <property name="idleTimeoutMinutes" value="600" />  
 <property name="matchOne" value="true" />  
 <property name="matchAll" value="true" />  
 <property name="selectOneAssumeMatch" value="true" />  
</bean>  
  
<bean id="connectionManager" class="org.jencks.factory.ConnectionManagerFactoryBean">  
 <property name="transactionSupport" ref="transactionSupport" />  
 <property name="poolingSupport" ref="poolingSupport" />  
</bean>   
  
<bean id="tranqlManagedConnectionFactory" class="org.jencks.tranql.XAPoolDataSourceMCF">  
 <property name="driverName"><value>com.sybase.jdbc2.jdbc.SybDriver</value></property>  
 <property name="url"><value>jdbc:sybase:Tds:recettemdp:4100/refstruct\_odyss\_dev?JCONNECT\_VERSION=5</value></property>  
 <property name="user"><value>xxx</value></property>  
 <property name="password"><value>xxx</value></property>  
</bean>  
  
<bean id="dataSourceRs" class="org.springframework.jca.support.LocalConnectionFactoryBean">  
 <property name="managedConnectionFactory"><ref local="tranqlManagedConnectionFactory"/></property>  
 <property name="connectionManager"><ref local="connectionManager"/></property>  
</bean>

Le datasource peut ensuite être injecté comme n'importe comme quelle dépendance, typiquement dans une session factory Hibernate.

On peut ensuite définir un service qui sera rendu transactionnel avec les définitions précédentes :

<bean id="service" class="fr.mediapost.municipost.spring.impl.ServiceDistribue">  
 <!-- injection de DAO utilisant les datasources définis avec Jencks -->  
 ...  
</bean>

Un souci que l'on peut rencontrer en utilisant les transactions distribuées avec Jencks est un manque de clarté sur les messages des exceptions, notamment les exceptions JDBC. Il est donc conseillé de toujours bien tester la couche de DAO et si possible en utilisant un gestionnaire de transaction plus adapté à la simplicité du scénario (un seul datasource, donc HibernateTransactionManager pour les DAO Hibernate et DataSourceTransactionManager pour les DAO JDBC), via un fichier Spring spécifique aux tests unitaires.

### Intercepteurs

Le framework AOP de Spring permet d'ajouter des comportements de façon complètement découplés à des objets. Dans une application J2EE, les objets susceptibles de profiter de ce système sont les classes de services et de DAO. Il est possible de choisir *quand* on veut faire intervenir un aspect (il s'agit de la notion d*'Advice*).

Une application intéressante est d'intercepter certains types d'exception lancés par les services et les DAO (voir aussi [la gestion des exceptions](http://sunic.mediapost.fr/dsiwiki/index.php?n=Transverse.Exceptions) pour des informations complémentaires). Par exemple, les DAO sont susceptibles de lancer des exceptions Hibernate. Or ce type d'exception ne doit en aucun cas être "catché" dans les couches supérieures, au risque de lier celles-ci à l'implémentation de la couche persistance (alors qu'il s'agit du but d'une couche de DAO).

Une solution consisterait à coder toutes les méthodes des DAO Hibernate de la façon suivante :

public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {  
  
 public Collection loadProductsByCategory(String category)  
 throws DataAccessException, MyException {  
  
 Session session = getSession(getSessionFactory(), false);  
 try {  
 List result = session.find(  
 "from test.Product product where product.category=?",  
 category, Hibernate.STRING);  
 if (result == null) {  
 throw new MyException("invalid search result");  
 }  
 return result;  
 }  
 catch (HibernateException ex) {  
 throw convertHibernateAccessException(ex);  
 }  
 }  
}

Avec la méthode convertHibernateAccessException de HibernateDaoSupport, on convertit toutes les exceptions Hibernate et aucune ne remonte dans les couches supérieures. Cette approche présente cependant des désavantages : code spaghetti avec le bloc try/catch, solution non-centralisée. **Cette approche est donc à proscrire**.

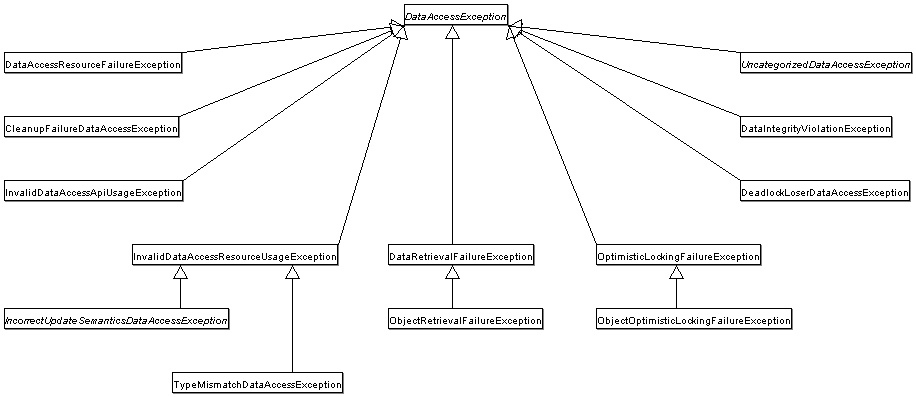
Avec de l'AOP, il est possible d'intercepter toutes les exceptions Hibernate et de les relancer, par exemple sous la forme d'une famille d'exceptions génériques. Un intercepteur serait donc comme suit :

package fr.mediapost.municipost.aop;  
  
import org.hibernate.HibernateException;  
import org.springframework.orm.hibernate3.SessionFactoryUtils;  
  
public class ExceptionIntercepteur {  
  
 public void handleException(HibernateException ex) {  
 throw SessionFactoryUtils.convertHibernateAccessException(ex);  
 }  
  
}

Il utilise un utiliaire Spring pour convertir les exceptions Hibernate. La configuration XML se fait de la manière suivante :

<!-- intercepteur relançant les exceptions -->  
<bean id="exceptionIntercepteur" class="fr.mediapost.municipost.aop.ExceptionIntercepteur" />  
  
<aop:config>   
 <!-- définition des méthodes des DAO -->  
 <aop:pointcut id="daoMethodes" expression="execution(\* fr.mediapost.municipost.integration..\*DAO\*.\*(..))"/>  
 <!-- relancer les exceptions -->  
 <aop:aspect id="aspectExceptions" ref="exceptionIntercepteur">  
 <aop:after-throwing pointcut-ref="daoMethodes" method="handleException" throwing="ex"/>  
 </aop:aspect>   
</aop:config>

Avec cette configuration, un proxy est automatiquement créé pour chaque DAO défini et les exceptions Hibernate sont systématiquement converties. L'arborescence d'exceptions utilisée est la suivante :



### Cycle de vie des Bean créés par Spring

#### Méthodes de callback

Il est possible de gérer le cycle de vie des Beans créés par Spring. On peut par exemple demander au conteneur Spring d'appeler une méthode à la création d'un Bean et une autre méthode à sa "destruction" (i.e. quand le conteneur est détruit). Prenons l'exemple suivant :

<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

Le bean défini doit donc disposer d'une méthode init qui sera appelée par le conteneur Spring juste après la création de l'objet via le constructeur par défaut :

public class ExampleBean {  
  
 public void init() {  
 // initialisation  
 }  
}

Pour éviter de préciser la méthode d'initialisation dans le fichier XML, il suffit de faire implémenter l'interface InitializingBean à la classe et le conteneur appellera automatiquement la méthode d'initialisation correspondante. Dans le fichier XML :

<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

Et dans la classe Java :

public class AnotherExampleBean implements InitializingBean {  
  
 public void afterPropertiesSet() {  
 // initialisation  
 }  
}

L'inconvénient de cette méthode est de lier le Bean à l'API Spring.

Il existe exactement le même principe avec des méthodes de destruction. En précisant dans le fichier XML :

<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>

Et dans la classe Java :

public class ExampleBean {  
  
 public void cleanup() {  
 // nettoyage, par exemple, fermer des connexions base de données  
 }  
}

Avec une interface Spring, dans le fichier XML :

<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

Et dans la classe Java (avec le même inconvénient de dépendance vis-à-vis de l'API Spring) :

public class AnotherExampleBean implements DisposableBean {  
  
 public void destroy() {  
 // nettoyage, par exemple, fermer des connexions base de données  
 }  
}

#### Evénements de l'Application Context

Un Application Context peut produire des événements le long de sa vie, typiquement une fois son chargement effectué ou lors de sa destruction. Qui dit production d'événements, dit écouteurs (en anglais *listeners*), c'est-à-dire des objets intéressés par ces événements. En implémentant l'interface ApplicationListener, un Bean défini dans un contexte Spring se voit enregistré comme écouteur des événements du contexte. Il sera donc prévenu lorsque des événements seront publiés sur le "canal" de l'Application Context. Il existe des événements par défaut, dont les plus intéressants sont ContextRefreshedEvent (chargement ou rafraichissement du contexte) et ContextClosedEvent (lors de la fermeture du contexte). Le système d'événement est extensible et en allant plus loin dans l'utilisation de l'API Spring, il est possible d'avoir des Bean générateurs de leur propre arborescence d'événements et donc des Bean y réagissant.

En restant sur les événements par défaut qui préviennent du cycle de vie du contexte, il est possible par exemple de faire des injections post-initialisation et donc de remédier à des problèmes de dépendances cycliques non-gérées par Spring. Prenons l'exemple suivant : un listener Hibernate doit effectuer des vérifications avant l'insertion en base de certains éléments. Ces vérifications ne sont pas codées en dur, elles font appel à un service métier. Or le listener doit être injecté dans la Session Factory Hibernate, elle-même injectée dans les DAO Hibernate et eux-mêmes dans des services métiers. On a donc typiquement une dépendance cyclique listener - session factory - dao - service métier - listener. Spring n'est pas capable de résoudre cette dépendance. Or, le listener Hibernate n'a pas besoin du service métier pour s'initialiser, on peut donc injecter celui-ci après l'initialisation totale du contexte. Un bon moyen de faire cela est d'utiliser le système d'événements de Spring : le listener peut implémenter l'interface ApplicationListener et récupérer le service dont il a besoin après l'initialisation du contexte.

##### Utilisation du système d'événéments pour de la post-injection

L'exemple précédent concernant le listener Hibernate nous a sensibilisés au problème de post injection, qui peut être utile dans le cas de dépendances cycliques. Le listener peut donc implémenter l'interface ApplicationListener et procéder de la manière suivante pour récupérer le service qui l'intéresse :

public void onApplicationEvent(ApplicationEvent baseEvt) {  
 if(baseEvt instanceof ContextRefreshedEvent) {  
 ContextRefreshedEvent evt = (ContextRefreshedEvent) baseEvt;  
 IMonService service = (IMonService) evt.getApplicationContext().getBean("monService");  
 setMonService(service)  
 }  
}

Cette méthode est intéressante mais on peut la rendre plus générique, par exemple en définissant une interface IPostInjectionNeeded :

/\*\*  
 \* Permet de configurer l'injection d'un bean défini dans un contexte Spring   
 \*/  
public interface IPostInjectionNeeded {  
  
 /\*\*  
 \* Le nom du bean à injecter  
 \* @return  
 \*/  
 public String getBeanName();  
  
 /\*\*  
 \* La propriété dans l'objet courant où doit être injecté le bean  
 \* @return  
 \*/  
 public String getPropertyName();  
  
}

Avec ses méthodes, l'interface définit une dépendance (le nom du Bean dans le contexte et le nom de la propriété dans le bean qui en dépend, un listener Hibernate dans notre exemple). On peut ensuite écrire une classe de listener dont une instance sera appelée à la fin du démarrage du contexte. Ce listener se chargera alors de récupérer tous les objets du contexte implémentant l'interface IPostInjectionNeeded et de leur injecter leur dépendance, grâce aux informations fournies par les 2 méthodes. Ce listener est défini de la sorte :

/\*\*  
 \* Bean se chargeant de la post-injection pour les <code>IPostInjectionNeeded</code> d'un contexte Spring.  
 \*/  
public class PostInjectionBean implements ApplicationListener {  
  
 private static final Log log = LogFactory.getLog(PostInjectionBean.class);  
  
 public void onApplicationEvent(ApplicationEvent baseEvt) {  
 if(baseEvt instanceof ContextRefreshedEvent) {  
 ContextRefreshedEvent evt = (ContextRefreshedEvent) baseEvt;  
 // récupération de tous les beans nécessitant une post injection  
 Map beans = evt.getApplicationContext().getBeansOfType(IPostInjectionNeeded.class);  
 // parcours des beans  
 for(Object key : beans.keySet()) {   
 IPostInjectionNeeded bean = (IPostInjectionNeeded) beans.get(key);  
 // on récupère la dépendance à injection dans l'application context  
 Object beanToInject = evt.getApplicationContext().getBean(bean.getBeanName());   
 try {  
 // on l'injecte  
 PropertyUtils.setProperty(bean,bean.getPropertyName(),beanToInject);  
 } catch (IllegalAccessException e) {  
 log.warn("Impossible de post injecter le bean "+bean.getBeanName()+" au bean "+key,e);  
 } catch (InvocationTargetException e) {  
 log.warn("Impossible de post injecter le bean "+bean.getBeanName()+" au bean "+key,e);  
 } catch (NoSuchMethodException e) {  
 log.warn("Impossible de post injecter le bean "+bean.getBeanName()+" au bean "+key,e);   
 }   
 }  
 }  
 }  
}

Il utilise les utilitaires d'introspection de Jakarta Commons BeanUtils pour effectuer l'injection. Ce listener du contexte Spring a juste à être déclaré de la manière suivante dans le contexte :

<bean id="postInjectionBean" class="fr.mediapost.municipost.spring.event.PostInjectionBean" />

Le listener Hibernate, objet de l'injection, aurait alors l'apparence suivante :

public class MonListener implements PreInsertEventListener, IPostInjectionNeeded {   
  
 private IMonService monService;  
  
 public boolean onPreInsert(PreInsertEvent evt) {   
 // fait qq chose en utilisant le service...  
 }  
  
 public IMonService getMonService() {  
 return monService;  
 }  
  
 public void setMonService(IMonService monService) {  
 this.monService= monService;  
 }  
  
 /\* implementation de l'interface IPostInjectionNeeded \*/  
 private String beanName;   
  
 /\*\* nom du bean (dans l'application context) à injecter \*/  
 public String getBeanName() {  
 return beanName;  
 }  
  
 public void setBeanName(Sting beanName) {  
 this.beanName = beanName;  
 }  
  
 /\*\* nom de la propriété dans ce bean \*/  
 public String getPropertyName() {  
 return "monService";  
 }  
  
}

La déclaration de l'application context serait comme suit :

<!-- le service métier -->  
<bean id="unServiceMetier" class="application.service.impl.MonServiceMetier">  
 <!-- injection de nombreux DAO -->  
 ...  
</bean>  
  
  
<!-- le listener Hibernate -->  
<bean id="monListenerHibernate" class="fr.mediapost.municipost.integration.hibernate.utils.event.MonListener">  
 <!-- on précise le nom du bean à post injection -->  
 <property name="beanName" value="unServiceMetier" />  
</bean>

Avec la déclaration du listener Spring, le listener Hibernate se verra injecter le service métier après l'initialisation complète de l'application context. L'exemple complet, basé sur un listener Hibernate vérifiant l'unicité d'un enregistrement avant insertion, est implémenté dans le prototype MunicipostPos.

## Validation

### Qu'est-ce qu'on entend par validation ?

La validation consiste à vérifier les données entrées par l'utilisateur, par exemple suite à la soumission d'un formulaire HTML. On peut distinguer 3 niveaux de validation :

1. syntaxique : les champs d'un formulaire étant toujours des chaînes de caractères, il faut veiller à ce que les types de champs spécifiques (dates, nombres) puissent être convertis correctement
2. métier simple : il s'agit de vérifications relativement simples, mettant en jeu la plupart du temps un seul objet. On peut citer par exemple : champ obligatoire, longueur maximum d'une chaîne de caractère, la date de début doit être *avant* la date de fin...
3. métier avancée : ces vérifications mettent en jeu des notions avancées du domaine fonctionnel, sur plusieurs objets et peuvent nécessiter de récupérer des données dans l'entrepôt de données

### Les difficultés de la validation

La validation est un sujet délicat. Un bon système de validation doit avoir les qualités suivantes :

* centralisé
* facile à tester
* précis (donner des messages d'informations clairs, quel que soit le contexte)
* être internationalisé

Une des principales difficultés est de savoir *où* faire la validation. Un piège fréquent est de concentrer la validation dans la partie présentation (dans des backing bean JSF ou dans des actions Struts). Cela nuit fortement à la réutilisabilité et à la testabilité de la validation. En effet, selon la taille d'une application et le nombre d'applications développées sur un même socle métier, il y a de fortes chances d'avoir besoin à plusieurs endroits des mêmes vérifications. D'où la nécessité de disposer d'une validation métier réutilisable et bien testée.

### Distribuer la validation dans les couches

La question est ici de savoir *où* faire la validation. La validation de niveau 1 est faite dans la couche présentation, en s'appuyant sur les fonctionnalités du framework (dans notre cas JSF). La validation de niveau 3 doit être concentrée dans la partie métier, dans des services spécifiques, appelés *Checker*. Ces objets peuvent être transactionnels ou pas, selon qu'ils accèdent ou non à la base de données.

Le niveau 2 de validation est le plus délicat à situer. On peut vouloir profiter des avantages de validation d'un framework tel que JSF et donc effectuer cette validation dans la couche présentation :

* existence de validateurs pré-définis
* bien intégré à la présentation (erreurs affichés facilement à coté des champs invalides)
* internationalisé

On a en revanche les inconvénients suivants :

* moins réutilisable
* validation disséminée donc un modèle métier moins autonome

On peut choisir de situer le niveau de validation dans la partie métier. Le mieux est de faire cette validation par entité, c'est-à-dire qu'une classe d'entité peut effectuer sa propre validation sur ses champs voire même sur les éléments qui dépendent d'elle. On a alors les avantages suivants :

* modèle métier consistant, autonome (une approche orientée objet)
* bonne testabilité (en général les classes d'entités sont des POJO et donc n'ont besoin d'aucun contexte pour être testés)
* évolutif et plus facilement maintenable (l'ajout d'une règle métier se fait à un endroit et est répercutée sur toutes les applications)

On a en revanche les inconvénients suivants :

* générique et donc pas toujours adapté à tous les contextes
* s'intègre moins bien au système de validation du framework de présentation
* technique : il faut implémenter des choses comme l'internationalisation, ajouter des annotations, passer par de l'AOP dans certains

Le choix pour le niveau 2 de validation est délicat à faire. Il est cependant possible de créer un framework de validation qui n'excluera aucune des deux politiques (dans la partie présentation ou tout dans la partie métier).

### Système de validation de la partie métier

Le système de validation métier est basé d'une part sur un ensemble de service, appelés Checkers et d'autre part sur une validation faite par les objets métier eux-mêmes.

#### Checkers

On peut dégager une interface simple et générique pour les checkers, afin de les manipuler le plus uniformément possible

public interface IChecker {  
  
 /\*\*  
 \* Effectue la validation.  
 \* @param object : typiquement un objet métier ou un DTO  
 \* @return une liste de message de validation (classe applicative)  
 \*/  
 public Collection <ValidationMessage> validate(Object object);  
  
}

Cependant, il est possible de créer des checkers avec des interfaces plus spécifiques selon les besoins. En poursuivant sur cette interface relativement simple, on peut écrire des implémentations qui s'avèrent efficaces pour valider un objet métier.

#### Validation par les objets métier

L'interface suivante peut être implémentée par une classe qui est susceptible de se valider :

/\*\*  
 \* Interface de validation.  
 \* Typiquement implémentée par un objet de domaine pour gérer sa propre validation.  
 \* @author Arnaud Cogoluegnes  
 \* @version $Id: IValidable.java,v 1.1 2006/09/19 16:00:52 acogolue Exp $  
 \*/  
public interface IValidable {  
  
 /\*\*  
 \* Méthode de validation.  
 \* @return  
 \*/  
 public Collection<ValidationMessage> validate();  
  
 /\*\*  
 \* Gestion de l'i18n.  
 \* @param msgSource  
 \*/  
 public void setMessageSource(MessageSource msgSource);  
  
}

On insiste fortement sur l'internationalisation en imposant l'injection d'une classe de MessageSource. Cela présente l'inconvénient de lier les POJO à l'API du MessageSource (Spring en l'occurence). Une contrainte technique est aussi d'injecter d'une manière ou d'une autre ce MessageSource.

Une classe de POJO peut alors se valider (le fonctionnel n'est pas vraiment réaliste dans cet exemple) :

public Collection<ValidationMessage> validate() {  
 Collection <ValidationMessage> res = new ArrayList<ValidationMessage>();  
 // vérification sur SIRET  
 if(getCliSiret() == null || !getCliSiret().startsWith("AA")) {  
 ValidationMessage msg = new ValidationMessage(  
 "cliSiret",  
 msgSource.getMessage("error.siret.code.incorrect",null,null)   
 );  
 res.add(msg);  
 }  
 return res;  
}

Il est judicieux d'effectuer dans cette méthode validate les validations portant sur plusieurs champs, par exemple la comparaison entre une date d'arrivée et une date de départ.

#### Réalisation de la validation d'un objet métier

##### Checker pour les IValidable

Il est maintenant possible d'assembler les notions de Checker et de IValidable :

/\*\*  
 \* Checker validant les objets métier implémentant IValidable  
 \* @author Arnaud Cogoluegnes  
 \* @version $Id$  
 \*/  
public class ValidableDomainValidator implements IChecker {  
  
 /\*\* i18n qui sera injecter aux objets domaines \*/  
 private MessageSource messageSource;  
  
 /\*\*  
 \* Validation  
 \*/  
 public Collection<ValidationMessage> validate(Object domainObject) {   
 if(domainObject instanceof IValidable) {  
 IValidable validable = (IValidable) domainObject;  
 validable.setMessageSource(getMessageSource());  
 return validable.validate();  
 } else {  
 return new ArrayList<ValidationMessage> () ;  
 }   
 }  
  
 public MessageSource getMessageSource() {  
 return messageSource;  
 }  
  
 public void setMessageSource(MessageSource messageSource) {  
 this.messageSource = messageSource;  
 }  
  
}

On remarque que le checker effectue l'injection du MessageSource dans l'objet métier, apportant facilement le support i18n.

##### Checker Hibernate Validator

Voici un exemple d'implémentation de checker utilisant Hibernate Validator :

/\*\*  
 \* Checker d'objets métier basé sur Hibernate Validator.  
 \* @author Arnaud Cogoluegnes  
 \* @version $Id$  
 \*/  
public class HibernateDomainValidator implements IChecker {  
  
 /\*\*  
 \* Validation  
 \*/  
 public Collection<ValidationMessage> validate(Object domainObject) {  
 Collection<ValidationMessage> messages = new ArrayList<ValidationMessage> ();  
 // création du validateur Hibernate  
 ClassValidator validator = new ClassValidator(domainObject.getClass());  
 // validation  
 InvalidValue[] validationMessages = validator.getInvalidValues(domainObject);  
 // transformation en messages applicatifs  
 for(int i=0,n=validationMessages.length;i<n;i++) {  
 ValidationMessage message = new ValidationMessage(  
 validationMessages[i].getPropertyName(),validationMessages[i].getMessage()  
 );  
 messages.add(message);  
 }  
 return messages;  
 }  
  
}

Ce checker renverra des erreurs selon les annotations qui ont été mises sur la classe de l'objet métier. Par exemple :

@Required(label = "Localité")  
@LongueurMax(label = "Localité", longueur = 26)  
private String adrLocalite;

On voit toutefois un inconvénient : l'internationalisation n'est pas gérée (des contraintes techniques la rendent difficile à implémenter).

##### Checker composite

Un checker composite permet d'assembler des checkers unitaires :

/\*\*  
 \* Checker composite.  
 \* @author Arnaud Cogoluegnes  
 \* @version $Id$  
 \*/  
public class CompositeCheckerValidator implements IChecker {  
  
 /\*\* liste des checkers unitaires \*/  
 private Collection <IChecker> checkers = new ArrayList <IChecker>();  
  
 /\*\*  
 \* Lance la validation des checkers unitaires.  
 \*/  
 public Collection<ValidationMessage> validate(Object domainObject) {  
 Collection<ValidationMessage> messages = new ArrayList<ValidationMessage> ();  
 for(IChecker validator : checkers) {  
 Collection <ValidationMessage> localMessages = validator.validate(domainObject);  
 // des erreurs de validation ?  
 if(localMessages != null && localMessages.size() > 0) {  
 messages.addAll(localMessages);  
 // on choisit d'arrêter la vérification s'il y a des erreurs  
 break;  
 }  
 }  
 return messages;  
 }  
  
 /\*\*  
 \* Ajout d'un checker.  
 \* @param validator  
 \*/  
 public void addChecker(IChecker validator) {  
 checkers.add(validator);  
 }  
  
 public Collection<IChecker> getCheckers() {  
 return checkers;  
 }  
  
 public void setCheckers(Collection<IChecker> checkers) {  
 this.checkers = checkers;  
 }  
}

On peut par exemple ajouter les 2 implémentations de checkers précédentes (Hibernate et IValidable) au checker composite et faire un unique appel dessus.

##### Assemblage avec Spring

Voici comment assembler les checkers et contrôler ainsi la validation de façon déclarative :

<!-- i18n -->  
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">  
 <property name="basenames">  
 <list>  
 <value>messages</value>  
 </list>  
 </property>  
</bean>  
  
<!-- validation -->  
<!-- checker composite pour les objets domaine -->  
<bean id="domainChecker" class="fr.mediapost.fwk.service.validation.impl.CompositeCheckerValidator">  
 <property name="checkers">  
 <list>  
 <!-- validateur Hibernate -->  
 <bean class="fr.mediapost.fwk.service.validation.impl.HibernateDomainValidator" />  
 <!-- validateur pour les IValidables -->  
 <bean class="fr.mediapost.fwk.service.validation.impl.ValidableDomainValidator">  
 <!-- injection i18n -->  
 <property name="messageSource" ref="messageSource" />  
 </bean>  
 </list>  
 </property>   
</bean>  
  
<!-- business delegate (façade métier) -->  
<bean id="protoBd" class="fr.mediapost.municipost.bd.impl.ProtoBdImpl">  
 <!-- checker pour les objets métier -->  
 <property name="domainChecker" ref="domainChecker" />  
 <!-- service transactionnel -->  
 <property name="serviceClient" ref="serviceClient" />   
</bean>

On remarque que le checker est injecté à une façade métier. Ainsi, la validation n'est appelée ni dans la couche présentation, ni dans la couche service. Voici un exemple d'appel dans la façade métier :

public void saveNewClient(RcClient client, RcAdresse adresse) throws BusinessException {  
 // vérification  
 // validation du client  
 Collection <ValidationMessage> messages = getDomainChecker().validate(client);  
 // validation de l'adresse  
 messages.addAll(getDomainChecker().validate(adresse));  
  
 // des erreurs ?  
 if(!messages.isEmpty()) {  
 // création des erreurs fonctionnelles  
 CompositeBusinessException ex = new CompositeBusinessException();  
 for(ValidationMessage msg : messages) {  
 ex.addBusinessException(msg.getMessage());  
 }  
 throw ex;  
 } else {  
 // appel du service  
 serviceClient.saveNewClient(client,adresse);  
 }   
}

On remarque que la validation des objets métier est très générique et indépendante des implémentations utilisées (ex. : Hibernate Validator). Il est donc d'une part possible d'ajouter des checkers divers et variés à la façade, selon les cas d'utilisation et d'autre part de modifier le checker de domaine (supprimer la validation de Hibernate Validator si le même cas de validation est effectué en amont, dans la partie JSF par exemple). Il n'y a aucune répercussion sur le code de la façade.

### Validation dans la couche présentation avec JSF

Voici un exemple où la validation de niveau 2 se trouve dans la partie présentation :

<h:inputText value="#{clientBean.saisieLocalite}" id="localite" required="true">  
 <f:validateLength maximum="26" />  
</h:inputText>  
<h:message for="localite" />

On remarque l'avantage d'utiliser le système de validation de JSF avec la balise <h:message for="localite" />, qui affichera l'erreur juste à côté du champ de saisie. Si l'on choisit de faire toutes les validations (mono-champ, car il est difficile de faire de la validation multi-champs avec JSF 1.1) de niveau 2 de cette façon, il faut alors désactiver les checkers correspondant dans la façade métier.

### Que faire dans un backing bean JSF ?

Pour un cas d'utilisation typique, le code d'une méthode de soumission d'un formulaire a l'apparence suivante :

/\*\*  
\* Insère un client en base de données.  
\*  
\*/  
public String insererClient() {  
 IServiceClient serviceClient = getServiceFactory().getServiceClient();  
 // création et synchronisation des objets métier  
 ...  
 // destination, par défaut on reste sur la page courante  
 String destination = null;   
 // récupération de la façade métier  
 IProtoBd protoBd = getBdFactory().getProtoBd();   
 try {  
 // appel du service métier  
 protoBd.saveNewClient(client,adresse);  
 // pas d'erreur, on redirige vers la page de recherche  
 destination = "rechercheClient";  
 } catch(BusinessException e) {  
 // exception fonctionnelle, on sauvegarde les erreurs  
 saveBusinessException(e);   
 }  
 return destination;  
}

L'appel saveBusinessException(e) sauvegarde les messages de l'exception dans le contexte JSF. Il s'agit d'une méthode à capitaliser dans une classe de base pour les backing bean.

# Framework Mediapost

Cette section recense les éléments de capitalisation de Mediapost, autour de la plateforme JEE et des outils utilisés.

## Identification

Projets CVS : mediapostFwkJEE, mediapostFwkJEETest, mediapostFwkJEESecurite

Groupe Maven : fr.mediapost.fwk

Artifacts Maven : mediapostfwkjee, mediapostfwkjeeTest, mediapostfwkjeeSecurite

Dernière version Maven : 1.0-SNAPSHOT

## Pourquoi un framework ?

Sur un ensemble de projets bâtis sur la même architecture technique et manipulant des notions métier communes, il est évident qu'il est possible de capitaliser du code. C'est pour cela qu'un projet distinct a été créé pour contenir toutes les classes réutilisables.

Cela permet donc de réutiliser du code fiable (s'il a été correctement testé) mais aussi d'alléger les différentes applications.

## Que contient le framework ?

Le framework est un framework *technique*. Il n'a pas pour but d'être un ensemble de classe définissant un système cohérent, avec des points d'extension (comme le sont par exemple Struts, Tapestry ou JSF pour les frameworks Web). Il a plutôt vocation de proposer des éléments réutilisables, des classes utilitaires ou abstraites de base (pour les DAO, les backing-bean). Néanmoins, certaines parties, comme par exemple la validation avec le système de checker, peuvent constituer des "mini-framework", ayant pour vocation d'être étendus puis réutilisés au fur et à mesure des projets.

Voici pour l'instant ce que contient le framework :

* AOP : intercepteur pour la gestion des exceptions
* domaine : une interface ITrackable (pour le suivi des modifications d'un objet)
* exceptions (ex. : BusinessException)
* persistance : classes de DAO de base, intercepteurs/loader/générateur de PK Hibernate...
* service : framework de validation
* web : classe de base pour les backing bean, convertisseur JSF
* tests unitaires : classes de base pour les tests unitaires des différentes couches d'une application (DAO, service)
* sécurité : configuration de base et outils pour intégrer ACEGI

## Tests unitaires

**Les tests unitaires sont indispensables à l'écriture du framework**. Il faut que ses classes soient bien testées, car beaucoup de code risque d'en dépendre. Les tests unitaires aident dans un premier temps à l'écriture des classes, voire au design. Ils permettent dans un deuxième temps d'éviter les régressions en cas de correction de bug ou d'amélioration. Voir [Les Tests unitaires](#scroll-bookmark-413)

L'infrastructure des tests unitaires est mise en place sur le projet du framework, il ne faut donc pas hésiter à les écrire.

## Intégration à Maven

La plupart des applications Web s'appuieront sur le framework et auront une dépendance. Pour pouvoir développer une application sous Eclipse en utilisant les classes du framework, il suffit soit de rajouter le JAR au classpath du projet, soit d'ajouter le projet du framework au projet de l'application.

Si l'on veut faire une publication du framework dans le dépôt Maven, il suffit de lancer les commandes de packaging et d'installation de Maven :

mvn package  
mvn install

## Evolutions du framework

Le framework sera amené à changer en fonction des développements. Il faudra donc bien évidemment songer à son versionning (via CVS) et à gérer les bonnes dépendances de version sur les applications l'utilisant.

## POM parent

### Identification

Projet CVS : mediapost-core

Groupe Maven : fr.mediapost

Artifact Maven : mediapost-core

voir [l'héritage de POM](http://sunic.mediapost.fr/dsiwiki/index.php?n=Maven.PresentationGeneraleMaven#heritagepom)

### Utilité

Ce POM est utilisé pour centraliser la définition des dépendances de base, des profils, des rapports des projets Mediapost.

Pour qu'un projet l'utilise, il faut incorporer ces lignes suivantes dans son POM :

<parent>  
 <groupId>fr.mediapost</groupId>  
 <artifactId>mediapost-core</artifactId>  
 <version>1.0</version>  
</parent>

## Projet qualité

### Identification

Projet CVS : mediapost-quality

Groupe Maven : fr.mediapost

Artifact Maven : mediapost-quality

Voir [Vérification de qualité du code](http://sunic.mediapost.fr/dsiwiki/index.php?n=Outils.Qualite)

### Utilité

Ce projet est destiné à rassembler les éléments liés à la qualité des projets Mediapost. Il est donc susceptible de contenir des fichiers de règles pour Checkstyle ou PMD, des implémentations spécifiques de règles pour ces mêmes outils.

Pour, par exemple, utiliser un fichier de règles Checkstyle défini dans les ressources de ce projet :

<build>  
 <plugins>  
...  
  
<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-checkstyle-plugin</artifactId>  
 <!-- dependances pour les fichiers de regles -->  
 <dependencies>  
 <dependency>  
 <groupId>fr.mediapost</groupId>  
 <artifactId>mediapost-quality</artifactId>  
 <version>1.0</version>  
 <scope>compile</scope>  
 </dependency>  
 </dependencies>  
</plugin>  
  
...  
  
 </plugins>  
</build>  
  
<reporting>  
 <plugins>  
  
...  
<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-checkstyle-plugin</artifactId>  
 <configuration>  
 <!-- une ressource se trouvant le projet mediapost-quality -->  
 <configLocation>config/mediapost\_checks.xml</configLocation>  
 </configuration>  
</plugin>  
...  
  
 </plugins>  
</reporting>

# Tests Unitaires Java

## Tests de la couche persistance

### Présentation

DBUNIT est une extension de JUNIT. Il s'agit d'une librairie permettant de faire des tests unitaires dans le cadre d'applications mettant en oeuvre des bases de données. DBUNIT permet donc d’industrialiser les données pour les tests unitaires portant sur les classes de la couche persistance (DAO).

DBUNIT permet de:

* Alimenter la base de données dans un état connu à partir de données contenues dans un fichier XML (dataset) avant chaque test unitaire
* Avoir des méthodes utilitaires pour récupérer le contenu des tables, ceci afin de comparer les contenus de bases (contenu obtenu après appel d'un traitement vs contenu attendu). Ce mécanisme de comparaison est basé sur des *assertions*.

### Best practices

#### Utiliser une instance de BD par développeur

Il est préconisé que chaque développeur dispose de son instance de base de données pour effectuer ses tests unitaires avec DBUNIT. Pour chaque test, une ou plusieurs tables doivent être nettoyées puis alimentées par les jeux de données (fichiers xml). Il est donc plus pratique, pour éviter les conflits entre développeurs, que chacun travaille sur sa propre instance

#### Utiliser de multiple petits datasets

Il ne sert à rien pour un test unitaire, qui n'est pas rappelons-le un test de performance, d'enregistrer des datasets importants : quelques enregistrements suffisent pour le test unitaire associé, correspondant au cas d'utilisation à tester.

#### Création d'un jeu de test

Pour créer du jeu de test (c'est-à-dire exporter les données de la base originale vers des fichiers xml exploitables par DbUnit) nous avons mis à disposition un projet utilitaire d'exportation permettant de générer automatiquement et très facilement :

* un fichier dtd définissant la structure de la base de donnée.
* un fichier xml contenant les données de la base originale utiles pour les tests

Il est ensuite possible de modifier éventuellement le fichier xml généré pour :

**Un projet a été créé pour Mediapost. Il est récupéré via CVS : il s'agit du projet Utils**

Ainsi toute création de datasets est outillée grâce à ce projet utilitaire : l'exportation des données de la base vers des fichiers xml est ainsi grandement facilitée.

#### Comment utiliser l'outil d'exportation des données de Mediapost (Projet Utils)?

* 1/ Renseigner les propriétés du projet dans le fichier dbunit/resources/export.properties
* 2/ Puis lancer l'export via outil d'exportation de données via l'outil d'exportation de données

Avec le script ant : build\_dbunit.xml  
Trois targets sont disponibles:

* 3/ Récupérer le fichier xml / la dtd à la racine du projet de test

### Les problèmes des contraintes sur les clés étrangères?

DbUnit permet d'insérer, de mettre à jour des lignes de tables dans le même ordre que celui du dataset xml et de supprimer dans l'ordre inverse. Il est donc important d'ordonner les lignes des différentes tables dans les datasets xml correctement pour éviter les problèmes de violation de contraintes sur les clés étrangères.

Depuis sa version 2.1, DbUnit propose un DatabaseSequenceFilter permettant de déterminer automatiquement l'ordre des tables en fonction des informations sur leur clés étrangères. L'exemple suivant illustre comment utiliser cette classe pour exporter les datasets XML. Cette classe a été intégrée dans le projet Utils de Mediapost, lorsque vous lancez la target *export\_tables* du script ant build\_dbunit.xml :

IDatabaseConnection conn = new DatabaseConnection(jdbcConn);   
ITableFilter filter = new DatabaseSequenceFilter(conn);  
IDataSet dataset = new FilteredDataSet(filter, conn.createDataSet());  
FlatXmlDataSet.write(dataset, new File(fileName));

### Comment comparer des jeux de données?

Lorsque vous exportez dans un fichier XML des données de la base, ce fichier sera utilisé dans un test unitaire. Au sein du test, vous pouvez vouloir comparer deux états de la base : l'état de certaines tables après l'appel effectif de la méthode à tester avec l'état attendu (expected dataset) de ces mêmes tables. DBUNIT propose des fonctions pour opérer ce type d'assertions.

Voici un exemple :

public void testCreate()  
 {  
 …   
 try {  
 // appel de la méthode à tester  
 clientDao.create(client);  
 // Récupération de l'état actuel des données de la base après l'execution du code à tester  
 IDataSet databaseDataSet = getConnection().createDataSet();  
 ITable actualTable = databaseDataSet.getTable("stock");  
  
 // Chargement des données attendues depuis un fichier XML expected\_client.xml  
 IDataSet expectedDataSet = new FlatXmlDataSet(new File("expected\_client.xml"));  
 ITable expectedTable = expectedDataSet.getTable("client");  
  
 // Comparaison des données de la base actuelle et des données attendues  
 Assertion.assertEquals(actualTable, expectedTable);  
  
 } catch (Exception e) {  
 fail(e.getMessage());   
 }  
 }

Vous avez également la possibilité d'exclure ou d'inclure des colonnes pour la comparaison des datasets. Pour cela, utilisez les méthodes de la classe AbstractDAOTest du framework Mediapost:

/\*\*  
 \* Comparaison sur une table entre le contenu de base et le dataset envoyé en excluant certaines colonnes dans la comparaison  
 \* @param nomTable le nom de la table à comparer   
 \* @param dataSet la ressource pointant vers le dataset attendu  
 \* @param tableau des noms de colonnes (string) à exclure de la comparaison  
 \*/  
 protected void comparerTablesFiltreColonnesExclues(String nomTable,String dataSet, String[] nomColonnesExclues) {  
 this.comparerTablesFiltreColonnes(nomTable,dataSet,nomColonnesExclues,true);  
 }  
  
 /\*\*  
 \* Comparaison sur une table entre le contenu de base et le dataset envoyé en considérant que certaines colonnes dans la comparaison  
 \* @param nomTable le nom de la table à comparer  
 \* @param dataSet la ressource pointant vers le dataset attendu  
 \* @param tableau des noms de colonnes (string) à inclure dans la comparaison  
 \*/  
 protected void comparerTablesFiltreColonnesIncluses(String nomTable,String dataSet, String[] nomColonnesIncluses) {  
 this.comparerTablesFiltreColonnes(nomTable,dataSet,nomColonnesIncluses,false);  
 }

Ces méthodes peuvent très utiles pour exclure de la comparaison des champs de tracking (timestamp, etc.)

### Ecriture d'un test unitaire pour un DAO

L'écriture d'un test unitaire se base sur la classe abstraite du framework mediapost AbstractDAOTest. Il suffit alors de :

* écrire un test qui hérite de la classe de base
* définir des fichiers pour le contexte applicatif Spring
* définir un dataset

Par exemple :

public class PersonneDAOTest extends AbstractDAOTest {  
  
  
 public void testSelectAll() {   
 IPersonneDAO dao = (IPersonneDAO) getBean("personneDAO");  
 // 3 enregistrements en base  
 assertEquals(3,dao.selectAll().size());  
 }   
  
 public void testInsert() {  
 ...  
 // exemple de code pour comparer le contenu de la table  
 // personne à celui défini dans le dataset attendu  
 comparerTables("personne","/fr/mediapost/fwk/test/expected-dataset-personne-dao.xml");   
 }  
  
 /\*\* définition du contexte Spring \*/   
 @Override  
 protected String[] getConfigLocations() {  
 return new String [] {"/fr/mediapost/fwk/test/application-context-test-dao.xml"};  
 }  
  
 /\*\* définition du dataset qui injecte des données entre chaque méthode de test \*/  
 @Override  
 protected String getDataSetResource() {  
 return "/fr/mediapost/fwk/test/dataset-personne-dao.xml";  
 }  
  
}

Un exemple de test de DAO est disponible dans le prototype MunicipostPos. Il s'agit de la classe de test du DAO HibernateJuridiqueDAO.  
**Attention :**

* **Pour lancer ce test, vous devez démarrer la base de données HSQLDB**
* **Pour effectuer cette tâche : exécuter via votre explorer windows le .bat suivant:** /unittest/launchDb.bat

### Comment procéder pour tester une couche de DAO dans une application

On a vu que la classe de base de test unitaire des DAO simplifie le travail. Néanmoins, elle n'est pas réutilisable directement pour un projet. En effet, autant le dataset dépend d'un test, autant la configuration de Spring a beaucoup de chances d'être la même pour toute une application. Celle-ci définit :

* la connexion à la base de données (préconisation : une instance de base de données par développeur)
* les DAO (et certainement une fabrique de DAO)
* la gestion des transactions : dans une application, les transactions sont gérées au niveau des services mais pour les tests unitaires sur les DAO, il faut que cette gestion se fasse au niveau des DAO.

On voit que la configuration peut être longue et qu'il serait pénible de la dupliquer. Donc lorsqu'un projet démarre, il est essentiel d'écrire une configuration Spring spécifique pour les DAO et une classe de base pour les tests unitaires des DAO, héritant de AbstractDAOTest. Celle-ci ne surchargera que la méthode getConfigLocations() et pourra fournir quelques méthodes de base (récupération de la fabrique de DAO par exemple).

Au final, écrire un test unitaire sur un DAO reviendra seulement à écrire une classe héritant de la classe abstraite applicative et à fournir un dataset (voire un dataset attendu, selon les cas).

### La difficulté du test des insertions

A noter que pour réaliser les tests, nous avons surcharger la classe fr.mediapost.fwk.integration.hibernate.utils.NextSequenceGenerator dans le projet prototype MunicipostPos (ressources des tests : src/test/java) afin d'utiliser un générateur de clés primaires simples (basé sur le générateur assigned d'Hibernate). Ceci afin de ne pas devoir disposer d'une procédure stockée spécifique dans l'environnement de test (base de données test).

## Tests de la couche métier

### De l'intérêt des Mocks Objects...

Il s'agit de tester les services de la couche métier. Comment tester unitairement? Comment tester un service métier sans ses dépendances, notamment ses dépendances vers des objets de type DAO? En effet, nous voulons ici tester un service métier sans disposer d'un système de persistance, d'une base de données.

La solution va venir principalement du principe d'inversion de contrôle : chaque service n'est jamais lié directement à l'implémentation d'un DAO. Chaque service métier se voit injecter une DAOFactory et utilise cette DAOFactory pour appeler un service spécifique d'un DAO. Voici un extrait d'un service métier du prototype MunicpostPos:

public class ServiceClientDefaultImpl extends MunicipostAbstractService implements IServiceClient{  
  
 private DAOFactory daoFactory;  
...  
 /\*\*  
 \* recherche des clients par noms avec Pagination  
 \* @param aName : nom du client  
 \* @param page : numéro de la page pour la pagination  
 \*/  
 public List findClientByNameAndPage(String aName, int page) throws BusinessException{  
  
 List listClients = this.getDaoFactory().getClientDao().findClientByNameAndPage(aName,page);  
 if ((listClients == null) || (listClients.isEmpty()))  
 {  
 // création des erreurs fonctionnelles  
 BusinessException ex = new BusinessException(getBundle().getMessage("msg.exception.0001",null,null));  
 throw ex;  
 }  
 return listClients;  
 }  
}

Dans un contexte de test unitaire, il s'agit alors de ne pas injecter les mêmes implémentations de DAO au service métier que celles utilisées dans un contexte de production. Nous allons donc injecter, pour tester unitairement un service métier, des implémentations *postiches* ou *bouchons* de DAO. Pour disposer de ces implémentations postiches, nous allons utiliser un framework Mock Objects.

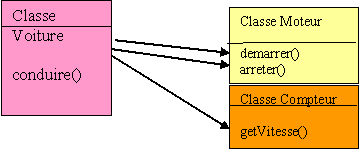
### Présentation des Mocks

Les mock objects sont des classes d’implémentation spécialement destinées à la réalisation de tests.

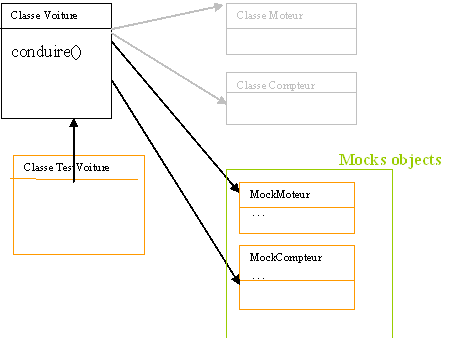
Un mock object est une implémentation d’une interface utilisée dans la classe que l’on veut tester.

### L'exemple naïf

Une classe Voiture utilise possède une méthode conduire() utilisant des méthodes de deux autres classes: Moteur et Compteur.



Pour tester de façon unitaire la méthode conduire() de notre classe Voiture qui utilise des méthodes contenues dans les classes Moteur et Compteur nous devons masquer le fonctionnement de ces dernières, car elles ne rentrent pas dans le domaine de notre test. On fonctionne en mode dit "boîte noire": des méthodes des classes Moteur et Compteur, on ne connaît que la signature, l'implémentation ne nous concerne pas. Pour masquer ces classes, nous allons leur substituer des classes factices : les mock objects.



De cette manière, on pourra tester complètement notre classe Voiture sans passer par Moteur et Compteur. Nos "mocks objects" vont se charger de renvoyer les valeurs attendues, récupérer des paramètres, lancer des exceptions que notre classe doit prendre en compte…, ceci par des méthodes génériques disponibles dans chaque mock object.

### Les Mocks dans le prototype

Tous les frameworks Mock proviennent d’un framework de base : voir <http://www.mockobjects.com>

Voici quelques exemples de frameworks Mock :

* MockMaker : <http://mockmaker.sourceforge.net>
* MockCreator : <http://mockcreator.sourceforge.net>
* JMock : <http://www.jmock.org>

Pour Mediapost, nous avons choisi le framework JMOCK. Nous préconisons vivement la lecture du [How TO JMOCK](http://sunic.mediapost.fr/dsiwiki/index.php?n=TestsUnitaires.HowToJmock)!!

Reprenons notre service métier ServiceClientDefaultImpl Ce service dispose d'une méthode findClientByNameAndPage permettant d'effectuer une recherche de clients (voir extrait de code ci-dessus). Cette méthode utilise un DAO ClientDao, pour effectuer cette recherche. Dans le contexte du test unitaire de cette méthode service, on va chercher à injecter non pas la réelle implémentation du DAO (une implémentation Hibernate en l'occurence) mais plutôt une implémentation postiche : un mock object. En faisant le choix de JMOCK, nous allons utiliser les principes des proxys pour générer ces mocks objects.

### JMOCK

Il s'agit d'un framework qui permet de ne pas générer manuellement les mocks objects pour les dépendances d'une classe à tester. La plupart des frameworks Mock Objects, type MockCreator, permettent effectivement de générer à la main, via l'IDE Eclipse (via en fait un plugin dédié) un mock à partir d'une interface. Typiquement, lorsque l'on veut tester un service métier, on va alors, avec MockCreator, générer un mock pour chaque DAO que ce service utilise, ceci à partir de son interface. Puis, on va définir un fichier de configuration de Spring dédié pour les tests unitaires de services, dans lequel on va configurer chaque service à tester. On va alors injecter à ces services les implémentations postiches de DAO générées précédemment.

Avec JMOCK, l'optique est différente. Il suffit dans la classe de test de spécifier que l'on veut mocker tel ou tel objet, ainsi l'étape manuelle consistant à générer les mocks, via l'IDE, n'est pas nécessaire. L'injection se fait programmatiquement, l'étape fastidieuse de déclaration de tous les mock objects dans un fichier de configurationde Spring n'est donc plus nécessaire

Voici un exemple de code illustrant un test simple de service métier, dans lequel on "mock" un DAO via JMOCK:

public class ServiceClientDefaultTest extends AbstractMunicipostServiceTest {  
  
 /\*\*  
 \* Test d'un appel OK  
 \*/  
 public void testfindRcJuridiqueByIdOK() throws Exception {  
  
 // Récupération du service via Spring  
 IServiceClient serviceClient = (IServiceClient) getBean("serviceClient");  
 long l = 9;  
  
 RcJuridique rcJuridique = new RcJuridique();  
  
 // on utilise JMOCK pour générer un proxy de RcJuridiqueDao  
 // on injecte alors le proxy généré dans la DAOFactory du service métier à tester   
 Mock mockdao = mock(IRcJuridiqueDAO.class);  
 serviceClient.getDaoFactory().setRcJuridiqueDao((IRcJuridiqueDAO) mockdao.proxy());  
  
 // on s'attend à ce que la méthode "findById" mock dao soit appelée une fois  
 // avec le parametre 'l' et retourne un objet de type rcJuridique  
 mockdao.expects(once())  
 .method("findById")  
 .with(eq(l)).will(  
 returnValue(rcJuridique));  
  
 // on lance l'exécution de la méthode à tester  
 serviceClient.findRcJuridiqueById(l);  
  
  
 }  
}

### Comment procéder pour tester une couche de service dans une application?

* Chaque test de ce type hérite d'une classe abstraite du type AbstractMonAppliServiceTest (voir prototype pour un exemple : AbstractMunicipostServiceTest). A noter que cette classe abstraite est générée lorsque vous utilisez les archetypes maven mediapost [voir la section Maven Archetypes](http://sunic.mediapost.fr/dsiwiki/index.php?n=Maven.ArchetypesMediapost). Vous devez, dans cette classe abstraite, simplement définir le fichier de configuration de Spring pour les tests, comme pour les tests de la couche persistance : voir [la section des tests des DAO](http://sunic.mediapost.fr/dsiwiki/index.php?n=TestsUnitaires.CouchePersistance#testdao). Vous devez, pour cela, surcharger la méthode protected String[] getConfigLocations() . Cette classe abstraite hérite d'une classe abstraite du framework JEE Mediapost fr.mediapost.fwk.test.AbstractServiceTestqui hérite elle-même d'une classe du framework JMock: MockObjectTestCase
* Vous pouvez ensuite tester les méthodes du service métier disposant d'une "logique métier propre". Nous préconisons de mettre l'effort de tests sur les méthodes de service métier encapsulant un algorithme métier particulier, et non pas sur les méthodes de service métier qui ne font qu'une déléguation simple à un service de persistance (encapsulant typiquement un appel à une méthode d'un DAO). Nous précisons ce point étant donné que l'exemple dans le prototype de test de service métier illustre un ou plusieurs tests pour des méthodes de service, simple ou complexe. Mais typiquement la méthode du service métier public void saveNewClient(RcClient aClient, RcAdresse adresse) n'a pas été testée, elle ne fait en effet qu'un appel à un DAO:

public void saveNewClient(RcClient aClient, RcAdresse adresse)  
 {  
 this.getDaoFactory().getClientDao().saveNewClient(aClient,adresse);  
  
 }

### JMOCK et tests avancés

#### Les exceptions des mocks

Nous avons vu un cas simple dans lequel on mockait un DAO et on s'attendait à ce qu'il retourne une valeur correcte. Essayons d'examiner ce que peut donner le même type de test unitaire en simulant que le DAO mocké renvoit une exception:

/\*\*  
 \* Test d'une exception levée par le DAO  
 \*/  
 public void testfindRcJuridiqueByIdKO() throws Exception {  
  
 IServiceClient serviceClient = (IServiceClient) getBean("serviceClient");  
 long l = 9;  
  
  
 Mock mockdao = mock(IRcJuridiqueDAO.class);  
 mockdao.expects(once())  
 .method("findById")  
 .with(eq(l)).will(  
 throwException(new RuntimeException("problème DAO") ) );  
  
  
 serviceClient.getDaoFactory().setRcJuridiqueDao((IRcJuridiqueDAO) mockdao.proxy());  
  
 try  
 {  
 serviceClient.findRcJuridiqueById(l);  
 fail ("doit catcher une exception");  
 }  
 catch (Exception e)  
 {  
 //OK  
 }  
  
 }

L'écriture du test reste simple : il suffit d'utiliser la méthode throwException de la classe mère du framework JMOCK : MockObjectTestCase. Dans notre exemple, lorsque le dao renvoit une exception, on vérifie que la méthode du service métier renvoie également une exception.

#### Un algorithme métier à tester impliquant plusieurs DAO

Voici un autre exemple : il s'agit de tester la méthode findClientByName de notre service métier permettant d'effectuer une recherche sur les clients. L'algorithme de cette méthode suppose que l'on effectue un premier appel au DAO (méthode countClient) pour nous donner le nombre de clients correspondant à nos critères de recherche, puis si ce nombre est inférieur ou égal à une certaine limite, on effectue la recherche en appelant une deuxième méthode du même DAO. Si le nombre renvoyé est supérieur à la limite, une BusinessException avec un message msg.exception.0002 indiquant qu'il est nécessaire de restreindre la recherche. Voici le code de ce service à tester:

/\*\*  
 \* Recherche des clients par noms sans Pagination  
 \* On vérifie si la liste est supérieure à NB\_LIMIT\_CLIENT  
 \* Si oui : BusinessException  
 \* Si non : on recupere de manière effective les elements clients  
 \* @param aName : nom du client  
 \* @param page : numéro de la page pour la pagination  
 \* @throws BusinessException si le nombre de client excede la limite  
 \* @return la liste des clients  
 \*/  
 public List findClientByName(String aName) throws BusinessException{  
  
 List listClients = null;  
 long nbclient = this.getDaoFactory().getClientDao().countClient(aName);  
 if (nbclient > this.getLimitNbClient())  
 {  
 // création d'une exception fonctionnelle  
 BusinessException ex = new BusinessException(getBundle().getMessage("msg.exception.0002",null,null));  
 throw ex;  
 }  
 else  
 {  
 listClients = this.getDaoFactory().getClientDao().findClientByNameAndPage(aName,0);  
 }  
 if ((listClients == null) || (listClients.isEmpty()))  
 {  
 // création d'une exception fonctionnelle  
 BusinessException ex = new BusinessException(getBundle().getMessage("msg.exception.0001",null,null));  
 throw ex;  
 }  
 return listClients;  
 }

En adéquation avec [nos préconisations](http://sunic.mediapost.fr/dsiwiki/index.php?n=TestsUnitaires.Methodologie), nous allons tester le service :

* dans son domaine de validité : nombre d'éléments inférieur à la taille limite
* hors de son domaine de validité : nombre d'éléments supérieur à la taille limite
* sur la limite de son domaine de validité : nombre d'éléments égal à la taille limite

Le premier test va consister à valider que si le premier appel du dao (on appelle une fois la méthode countClient) renvoit un nombre d'éléments **inférieur** à la taille limite, alors le deuxième appel du dao est effectué (on appelle une fois la méthode findClientByNameAndPage) :

/\*\*  
 \* Test de deux appels successifs OK d'un DAO : ClientDao  
 \* le premier appel de la méthode countClient renvoit un nombre d'elements inferieur a la taille limite  
 \*/  
 @SuppressWarnings("unchecked")  
 public void testFindClientByNameOK() throws Exception {  
  
 IServiceClient serviceClient = (IServiceClient) getBean("serviceClient");  
 long l = 0;  
  
 List listclient = new ArrayList();  
 // la liste des clients ne doit pas être vide : on ajoute un objet de type client  
 RcClient client = new RcClient();  
 listclient.add(client);  
  
  
 Mock mockdao = mock(IClientDAO.class);  
 serviceClient.getDaoFactory().setClientDao((IClientDAO) mockdao.proxy());  
  
 mockdao.expects(once())  
 .method("countClient")  
 .with(eq("jean")).will(  
 returnValue(l));  
  
 mockdao.expects(once())  
 .method("findClientByNameAndPage")  
 .with(eq("jean"),eq(0)).will(  
 returnValue(listclient));  
  
 serviceClient.findClientByName("jean");  
  
 }

Le deuxième test va consister à valider que si le premier appel du dao (on appelle une fois la méthode countClient) renvoie un nombre d'éléments **supérieur** à la taille limite, alors le deuxième appel du dao n'est pas effectué. Dans ce cas, la méthode du service doit également renvoyer une exception (on teste même le libellé du message de cette exception métier) :

/\*\*  
 \* Test d'un appel d'un DAO : ClientDao renvoit un nombre superieur a la limite :  
 \* l'appel de la méthode findClientByNameAndPage n'est donc pas fait  
 \* et une exception dont le message est 'msg.exception.0002' est levée  
 \*/  
 @SuppressWarnings("unchecked")  
 public void testFindClientByNameKO() throws Exception {  
  
 IServiceClient serviceClient = (IServiceClient) getBean("serviceClient");  
 // le nombre de clients retournés excède la limite  
 long l = serviceClient.getLimitNbClient()+1;  
  
 Mock mockdao = mock(IClientDAO.class);  
 serviceClient.getDaoFactory().setClientDao((IClientDAO) mockdao.proxy());  
  
 // on appelle ue fois le dao clientDAO : méthode countClient  
 mockdao.expects(once())  
 .method("countClient")  
 .with(eq("jean")).will(  
 returnValue(l));  
  
 try  
 {  
 serviceClient.findClientByName("jean");  
 fail ("doit catcher une exception");  
 }  
 catch (Exception e)  
 {  
 // on vérifie le message de l'exception  
 assertEquals(e.getMessage(),serviceClient.getBundle().getMessage("msg.exception.0002",null,null));  
 }  
  
 }

Le troisième test va consister à valider que si le premier appel du dao (on appelle une fois la méthode countClient) renvoie un nombre d'éléments **égal** à la taille limite, alors le deuxième appel du dao est bien effectué :

/\*\*  
 \* Test de deux appels successifs OK d'un DAO : ClientDao  
 \* le premier appel de la méthode countClient renvoit un nombre d'elements égal à la taille limite  
 \*/  
 @SuppressWarnings("unchecked")  
 public void testFindClientByNameLimite() throws Exception {  
  
 IServiceClient serviceClient = (IServiceClient) getBean("serviceClient");  
 // le nombre de clients retournés = la limite  
 long l = serviceClient.getLimitNbClient();  
  
 List listclient = new ArrayList();  
 // la liste des clients ne doit pas être vide : on ajoute un objet de type client  
 RcClient client = new RcClient();  
 listclient.add(client);  
  
  
 Mock mockdao = mock(IClientDAO.class);  
 serviceClient.getDaoFactory().setClientDao((IClientDAO) mockdao.proxy());  
  
 mockdao.expects(once())  
 .method("countClient")  
 .with(eq("jean")).will(  
 returnValue(l));  
  
 mockdao.expects(once())  
 .method("findClientByNameAndPage")  
 .with(eq("jean"),eq(0)).will(  
 returnValue(listclient));  
  
 serviceClient.findClientByName("jean");  
  
 }

## How to jMock

### Préconisations sur l'organisation de la méthode de test

Nous préconisions d'écrire un test avec JMOCK en respectant l'enchainement suivant:

* initier le test, générer les mocks et les injecter:

Mock mockdao = mock(IClientDAO.class);serviceClient.getDaoFactory().setClientDao((IClientDAO) mockdao.proxy());

* définir ce que l'on attend des mocks (**expectations** ou **stubs**), les contraintes (**with(eq(...))**), etc.:

mockdao.expects(once()).method("countClient").with(eq("jean")).will(returnValue(l));

* invoquer la méthode à tester :

serviceClient.findClientByName("jean");

* vérifier les **expectations** et utiliser si nécessaire les méthodes d'assertions pour vérifier les post-conditions du test:

mockdao.verify(); : cette méthode n'est pas à appeler explicitement dans les tests écrits pour Mediapost puisque nous avons préconisé et implémenté dans le framework et le prototype une solution pour qu'ils héritent tous de MockObjectTestCase (voir [Comment procéder pour tester une couche de service dans une application?](http://sunic.mediapost.fr/dsiwiki/index.php?n=TestsUnitaires.CoucheM%e9tier#testservicemetier))assert...

### Comment définir ce que l'on attend d'un Mock?

Une fois que l'on a généré un mock à partir d'une interface, l'intérêt est de pouvoir manipuler ce mock. On va pouvoir ainsi définir des contraintes sur les arguments passés en paramètres des méthodes invoquées sur ce mock. On peut également définir ce que l'on attend en retour de ces mêmes méthodes, etc.

La syntaxe pour positionner ces clauses peut paraître quelque peu obscure à la première utilisation. Voici la chaîne d'appel de ces clauses, chaque clause sera ensuite décrite.

Soit mockdao le mock généré à partir d'une interface.

mockdao.expects(expectation) ou stubs()  
 .method(nom de la methode) : possibilité de positionner des filtres du type par exemple : method(startingWith(...))  
 .with(contraintes) ou withAnyArguments() ou withNoArgument()  
 .after(id de la méthode invoquée antérieurement)  
 .match(autre règle spécifique définie pour ce cas de test)  
 .will(comportement attendu)  
 .id(id de cette invocation, utilisée ensuite potentiellement dans un after())

### Que sont les stubs et expectations JMOCK?

Dans Jmock, les stubs (exemple : stubs()) et expectations (exemple : expects(once()),expects(atLeastOnce())) sont presque la même chose : ils permettent d'initier l'écriture des clauses correspondant à ce que l'on attend d'un mock. Il s'agit de la première méthode appelée sur un mock.

Un stub permet simplement d'indiquer que l'on s'attend de zéro ou plusieurs invocations de méthodes du mock. on ne précise pas le nombre de fois où le mock va être invoqué, la syntaxe est donc plus simple, le test moins contraint.

Ainsi le code suivant où on indique que l'on s'attend à ce que le mockdao soit appelé 1 fois :

mockdao.expects(once())  
 .method("countClient")  
 .with(eq("jean")).will(  
 returnValue(l));

peut être remplacé par celui-ci où on ne précise pas le nombre d'invocation du mockdao :

mockdao.stubs()  
 .method("countClient")  
 .with(eq("jean")).will(  
 returnValue(l));

**Les valeurs possible de la méthode** expects(...) **sont**:

* once() : une seule fois
* atLeastOnce() : au moins une fois
* atMostOnce() : au plus une fois
* never() : jamais
* exactly(#nb) où #nb correspond au nombre exact d'invocations de la méthode. Par exemple once() est équivalent à exactly(1)

**A noter que les arguments de la méthode** expects(...) **sont des objets qui implémentent tous l'interface** InvocationMatcher **du framework jMock. Il est possible de définir ses propres implémentations.** Voir : <http://www.jmock.org/custom-matchers.html>

Pour spécifier le nom de méthode, il est possible soit d'indiquer le nom complet de la méthode :

mockdao.stubs()  
 .method("countClient")  
 ...

soit d'utiliser les filtres suivants pour indiquer que la méthode commence par, contient ou finit par une chaîne de caractères donnée :

mockdao.stubs()  
 .method(startsWith("count"))  
 ...  
 mockdao.stubs()  
 .method(stringContains("count"))  
 ...  
 mockdao.stubs()  
 .method(endsWith("count"))

Quand une méthode est invoquée sur un mock object, le mock object recherche à travers ses **expectations** depuis la plus récente jusqu'à la plus ancienne pour en trouver une qui correspond à l'invocation. Après l'invocation, l**'expectation** correspondante peut être ou non utilisée pour les invocations suivantes. Par exemple, une **expectation** de type expects(once()) ne sera utilisée qu'une seule fois pour la recherche de la correspondance et sera ignorée pour les futures invocations tandis qu'une **expectation** de type expects(atLeastOnce()) sera toujours utilisée pour cette recherche.

Attention:

* si on crée **une expectation puis un stub** pour la même méthode, le **stub** écrasera toujours l' **expectation** et celle-ci ne sera jamais utilisée.
* si on crée différentes **expectations** pour la même méthode, elles seront invoquées dans l'ordre opposé à celui selon lequel elles sont spécifiées.

La meilleure chose à faire est de ne pas utiliser de multiples **expectations et stubs** pour la même méthode avec exactement les mêmes contraintes d'appel. Utiliser la méthode onConsecutiveCalls pour créer de multiples actions pour une méthode. Par exemple:

mockdao.expects(atLeastOnce()).method(countClient).with(...)  
 .will( onConsecutiveCalls(  
 returnValue(10),  
 returnValue(20),  
 throwException(new IOException()) ) );

S'il est nécessaire de spécifier un ordre plus complexe sur l'invocation des mocks objects, il est conseillé d'utiliser la méthode after pour définir explicitement un ordre partiel ou complet des invocations. Par exemple:

mockdao.expects(once())  
 .method("countClient")  
 .with(eq("jean")).will(  
 returnValue(l)).id("premier appel à countClient");;  
  
 mockdao.expects(once())  
 .method("findClientByNameAndPage")  
 .with(eq("jean"),eq(0)).after("premier appel à countClient").will(  
 returnValue(listclient));

### Comment définir des contraintes sur les MOCKs ?

Lorsque des **expectations** sont définies sur des mock objects, jMock oblige à définir de manière explicite certains arguments aux méthodes de type expects() ou stub(). Les valeurs des arguments attendus sont définies à travers des **contraintes** fournies en paramètres des méthodes:

* with()
* withAnyArguments() : test très lâche, on ne vérifie pas les arguments
* withNoArgument(): test fixe, on vérifie qu'aucun argument n'est passé

jMock fournit un ensemble de contraintes qui pourront être utilisées dans la plupart des tests. Ces contraintes peuvent être combinées pour préciser ou assouplir les spécifications si nécessaire. L'ensemble de contraintes est extensible, il est possible en effet d'écrire ses propres contraintes pour couvrir des cas de tests spécifiques. Ces contraintes sont spécifiées en appelant des méthodes (eq(...), etc.) de la classe mère des tests unitaires utilisant jMock, à savoir : MockObjectTestCase

### Contraintes basiques

#### Egalité

La contrainte la plus utilisée est **eq**. Elle permet de spécifier la valeur des arguments. Le code ci-dessous indique par exemple que l'on s'attend à ce que la méthode findById du mockdao soit appelée une fois (méthode once()) avec (méthode with()) un argument de type long égal à 9 (utilisation de la contrainte eq(l))

long l = 9;  
  
 mockdao.expects(once())  
 .method("findById")  
 .with(eq(l)).will(  
 throwException(new RuntimeException("problème DAO") ) );

La contrainte **eq** utilise la méthode equals de la valeur attendue pour comparer les valeurs réelles et attendues. Les valeurs Null sont tout d'abord vérifiées, il peut donc être approprié de spécifier eq(null) ou d'appliquer une contrainte à une valeur réelle nulle.

La contrainte **eq** est surchargée pour tous les types primitifs (int, long, etc.). Les valeurs de type primitif sont ainsi wrappées dans des objets (Integer, Long, etc.) utilisés alors par la comparaison (via leurs méthodes equals()). Les tableaux sont traités de manière spécifique : deux tableaux sont considérés égaux par la méthode **eq** si et seulement si ils ont la même taille et que tous leurs élements sont considérés égaux par la méthode **eq**.

#### Egalité numérique avec une marge d'erreur

Une version surchargée de la contrainte **eq** permet de spécifier des valeurs flottantes pour encadrer la valeur attendue. Le code ci-dessous indique par exemple que l'on s'attend à ce que la méthode findById du mockdao soit appelée une fois (méthode once()) avec (méthode with()) un argument strictement supérieur à 8.997 et strictement inférieur à 9.003 :

mockdao.expects(once())  
 .method("findById")  
 .with(eq(9,0.003)).will(  
 throwException(new RuntimeException("problème DAO") ) );

#### Contrainte Anything

La contrainte **ANYTHING**  permet de spécifier que n'importe quelle valeur est acceptée. Attenton, il s'agit d'une constante et non d'une méthode. Cette clause peut être intéressante pour ignorer les arguments qui ne sont pas nécessaires pour le cas de test en cours. Les tests restent ainsi flexibles et plus indépendants de la classe à tester, la maintenance du test va ainsi être facilitée. Le code ci-dessous indique par exemple que l'on s'attend à ce que la méthode countClient du mockdao soit appelée une fois avec n'importe quel argument.

mockdao.expects(once())  
 .method("countClient")  
 .with(ANYTHING).will(  
 returnValue(l));

Si tous les arguments sont à ignorer, il convient alors d'utiliser la méthode : withAnyArguments()

#### Contraintes Nulles ou Non Nulles

Les contraintes **NULL** et **NOT\_NULL** permettent de spécifier respectivement que l'argument est null ou non null. Attention, il s'agit de constantes et non de méthodes. Le code ci-dessous indique par exemple que l'on s'attend à ce que la méthode countClient du mockdao soit appelée une fois avec un argument NON NULL.

mockdao.expects(once())  
 .method("countClient")  
 .with(NOT\_NULL).will(  
 returnValue(l));

#### Identité des objets

La contrainte **same** spécifie que la valeur réelle de l'argument et la valeur attendue correspondent au même objet. Il s'agit d'une contrainte très précise, plus fine que **eq** mais très utile : c'est en effet un cas classique que de vouloir passer des objets (entités ou DTO) en paramètres de certaines méthodes.

#### Vérification sur les types des instances

La contrainte **isA** permet de spécifier que l'argument réel doit être une instance du type donné. Le code ci-dessous indique par exemple que l'on s'attend à ce que la méthode countClient du mockdao soit appelée une fois avec un argument de type String:

mockdao.expects(once())  
 .method("countClient")  
 .with(isA(String.class)).will(  
 returnValue(l));

#### Les Strings

La contrainte **stringContains**  permet de spécifier que l'argument attendu doit être un objet de type String contenant une sous-chaîne donnée. Le code ci-dessous indique par exemple que l'on s'attend à ce que la méthode countClient du mockdao soit appelée une fois avec un argument de type String contenant la sous-chaîne "jean":

mockdao.expects(once())  
 .method("countClient")  
 .with(stringContains("jean")).will(  
 returnValue(l));

### Combiner les Contraintes

Les contraintes peuvent être combinées pour "affiner" ou inversement "relacher" la spécification. Les contraintes composites sont elles-mêmes des contraintes et obéissent aux mêmes règles : elles peuvent être également composées (plusieurs niveaux d'imbrication de contraintes Not/And/Or).

#### Not — Négation Logique

La contrainte **not** spécifie que l'argument ne doit pas correspondre à une contrainte donnée. Le code ci-dessous indique par exemple que l'on s'attend à ce que la méthode findById du mockdao soit appelée une fois avec un argument qui ne doit pas être égal à 2:

mockdao.expects(once())  
 .method("findById")  
 .with(not(eq(2))).will(  
 returnValue(rcJuridique));

#### Et — Conjonction logique

La contrainte **and**  spécifie que l'argument doit correspondre aux deux contraintes données. Le code ci-dessous indique par exemple que l'on s'attend à ce que la méthode findById du mockdao soit appelée une fois avec un argument qui ne doit pas être égal à 2 ET qui ne doit pas être égal à 3:

mockdao.expects(once())  
 .method("findById")  
 .with(and(not(eq(2)),not(eq(3)))).will(  
 returnValue(rcJuridique));

#### Or — Disjonction logique

La contrainte **or** spécifie que l'argument doit correspondre à l'une des deux contraintes données. Le code ci-dessous indique par exemple que l'on s'attend à ce que la méthode findById du mockdao soit appelée une fois avec un argument qui doit être égal à 9 OU qui doit être égal à 10:

mockdao.expects(once())  
 .method("findById")  
 .with(or(eq(9),eq(10))).will(  
 returnValue(rcJuridique));

### Nos propres contraintes

Il est parfois nécessaire de définir ses propres contraintes :

* soit en utilisant des contraintes existantes comme nous l'avons expliqué ci-dessus
* soit en étendant l'ensemble des contraintes jMock existantes : voir <http://jmock.codehaus.org/custom-constraints.html>

## Méthodologie de tests unitaires

### Le test unitaire doit :

* faire ressortir les défauts non prévus,
* contrôler les données invalides,
* contrôler les limites et les cas hors limites

### Que faut-il tester???

Il n'y a pas de règle : *"Everything that could possibly break"*.

Mais uniquement des principes:

* **Tester une méthode sans valeur de retour**: généralement, la méthode effectue quelque chose ! Elle a donc un effet de bord sur des objets de la classe fonctionnelle. Le test porte donc sur ces effets.
* **Tester un domaine de validité**: tout test portant sur la validité d'un domaine doit:
  + vérifier le comportement dans le domaine de validité
  + vérifier le comportement hors du domaine de validité
  + vérifier le comportement sur les valeurs limites
* **Tester des exceptions**: tester la levée d'une exception suit le même principe que pour tester un domaine de validité. Généralement d'ailleurs ce type de test est complètement lié au test sur le domaine de validité (typiquement lorsqu'on est hors du domaine, une exception est levée). Doivent être testés:
  + le cas où l'exception n'est pas levée
  + le cas où l'exception est levée.

### D'autres préconisations :

#### Tests par classe : une classe de test par classe???

Oui : pour toute classe non élémentaire. Pour distinguer une classe élémentaire : se poser la question « la classe a-t-elle une logique propre? ». Exemple de classes élémentaires : classes d’exceptions

#### Plusieurs méthodes de test pour une méthode à tester

Ainsi, lorsqu’un des cas de test échoue : pas de blocage pour les autres tests

#### Les objets du domaine : doit-on tester les fonctions getter et setter d’attributs?

Non : pas utile, on voit rapidement s’ils sont corrects. De plus ce type d'objets est bien souvent généré automatiquement par les IDE (Clic Droit dans Eclipse -> Source -> Generate Getters an Setters) Néanmoins, les objets du domaine encapsulent des méthodes de traitement (validate()) par exemple qui, elles, doivent être testées.

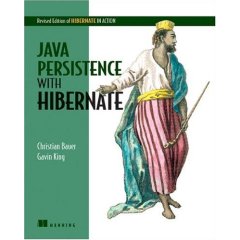
#### Les propriétés non publiques de l’objet doivent-elles être testées et les tests doivent-ils accéder aux méthodes & attributs non publics?

* Grand débat dans la communauté : voir <http://c2.com/cgi/wiki?UnitTestingNonPublicMemberFunctions>
* À la base, les tests unitaires vérifient les méthodes de l’interface de la classe, donc uniquement les méthodes publiques.
* Néanmoins, les méthodes privées peuvent être assez complexes et contenir des bugs.
* Ron Jeffries : « si l’on pense qu’un test important ne peut être réalisé qu’en accédant au détail de l’implémentation, il faut alors le faire. S’il s’avère que l’implémentation est trop instable, on s’en aperçoit très vite. Si malgré tout l’implémentation est stable, cela peut vouloir dire que l’élément non public est en réalité un élément public non identifié, qui cherche encore son client»
* Pour tester les méthodes non publiques : utiliser la *réflexion* (attention fastidieux) ou, soyons pragmatiques, on peut parfois s'autoriser à tester globalement deux méthodes (une méthode publique utilisant la méthode privée que l'on cherche à tester) à travers une méthode de test appelant la méthode publique. Effectivement ici le test ne sera pas réellement unitaire dans le sens où il portera sur deux méthodes, mais dans certains cas, ce choix peut s'avérer beaucoup plus simple que d'utiliser la réflexion.

# Bibliographie

Livres sur HIBERNATE :

* + Java Persistence with Hibernate (PDF : [bauer2\_ebook.pdf](http://appsiege.mediapost.fr:98/confluence/download/attachments/917710/bauer2_ebook.pdf?version=1&modificationDate=1335361788000&api=v2))



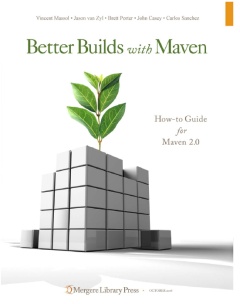
* Livres sur SPRING :
  + Spring in Action (PDF : [walls2\_ebook.pdf](http://appsiege.mediapost.fr:98/confluence/download/attachments/917710/walls2_ebook.pdf?version=1&modificationDate=1335361821000&api=v2))



* Livres sur JSF :
  + Java Server Faces in Action (PDF : [mann\_ebook.pdf](http://appsiege.mediapost.fr:98/confluence/download/attachments/917710/mann_ebook.pdf?version=1&modificationDate=1335361846000&api=v2))



* Livres sur MAVEN :
  + Better Builds with Maven (PDF : [BetterBuildsWithMaven.pdf](http://appsiege.mediapost.fr:98/confluence/download/attachments/917710/BetterBuildsWithMaven.pdf?version=1&modificationDate=1335361861000&api=v2))



* Documentation ACEGI Security
  + Documentation de référence de la version 1.0.3 (PDF : [acegi.pdf](http://appsiege.mediapost.fr:98/confluence/download/attachments/917710/acegi.pdf?version=1&modificationDate=1335361902000&api=v2))

