|  |
| --- |
| UNiversite de montpellier ii |
| Outil de C.M.S. sur la plateforme Seam |
|  |
| *Travail D’Etude et de Recherche effectué dans le cadre du master informatique 2ème année spécialité Génie Logiciel à la faculté des Sciences de l’Université de Montpellier II.* |
|  |
|  |

AHAMADA Soulé, DUVAL Simon, KEIRA Mamadou et TOUIL Mehdi

Tuteur s :

Mr DONY Christophe

Mr CHATEL Thierry

# Introduction

Dans le cadre de notre formation visant à obtenir le master informatique à vocation professionnel, nous avons effectué ce T.E.R. (Travail d’Etude et de Recherche) dont l’objectif initial était de concevoir un système de gestion de contenu, plus communément appelé CMS (Content Management System). Ce dernier est définit, selon Wikipédia, comme « une famille de logiciels destinés à la conception et à la mise à jour dynamique de site web ou d'application multimédia ».

En collaboration avec IOcean, entreprise située à Montpellier, nous avons déterminé le besoin fonctionnel du T.E.R.: Une première partie « arrière-boutique », que l’on appellera par la suite « Back Office », concernera l’administration et la gestion du site (gestion du contenu, publication, gestion des utilisateurs et des droits, etc.) ; une seconde partie, correspondant à l’interface de l’application que nous appellerons par la suite Front Office, affichera le contenu consultable.

Il est important de noter qu’il existe déjà un grand nombre d’implémentations de systèmes de gestion de contenu. L’objectif du T.E.R. n’est donc pas de créer une implémentation de plus, mais bien d’apporter de nouvelles idées. Celles-ci porteront sur la manière de gérer le contenu, les utilisateurs et les droits qu’auront ces derniers sur le contenu.

L’application a été développée sur la plateforme JBoss Seam (topo Seam). Elle implémente les fonctionnalités principales que requièrent un CMS et, évidemment, nos propres idées. Certaines fonctionnalités qui n’ont pas été implémentées en raison du temps impartie au T.E.R. seront tout de même expliquées.

Ce mémoire vous informera, dans un premier temps, sur la méthode de travail utilisée, ainsi que la gestion du projet en général ; dans un deuxième temps nous présenterons l’analyse en expliquant le choix des solutions retenues ainsi que celles délaissées ; nous continuerons sur la conception et l’implémentation du logiciel en pointant notre documentation et explication sur le fonctionnement de la gestion du contenu et des droits utilisateurs. Pour finir nous établirons un bilan sur le résultat obtenu, sur les difficultés rencontrées, et sur l’apport personnel de ce T.E.R.

# Méthodologie de travail

## Gestion des ressources

Notre groupe de T.E.R. est composé de 4 membres : Ahamada Soulé, Duval Simon, Keira Mamadou et Touil Mehdi. Ce dernier n'a pu participé à la réalisation de ce projet du à des problèmes de santé.

Lors de notre première réunion nous avons décidé de gérer ensemble la partie analyse. Il nous semblait important que chacun puisse, d’une part, apporter des idées sur l’architecture du projet et, d’autre part, avoir une vue d’ensemble sur la solution retenu pour la réalisation du projet. Ensuite, pour la partie conception et la rédaction de ce rapport, nous avons préféré nous partager le travail de manière à progresser plus rapidement dans la réalisation.

## Gestion de projet

Après la lecture du cahier des charges, lors de l’analyse, nous avons effectué le découpage du projet et conçut une planification prévisionnelle du projet à l'aide de l'outil Gantt (voir Figure 1) en tenant compte des ressources et du temps disponible. Ce planning nous a servi essentiellement au respect des délais.

### Planning prévisionnel

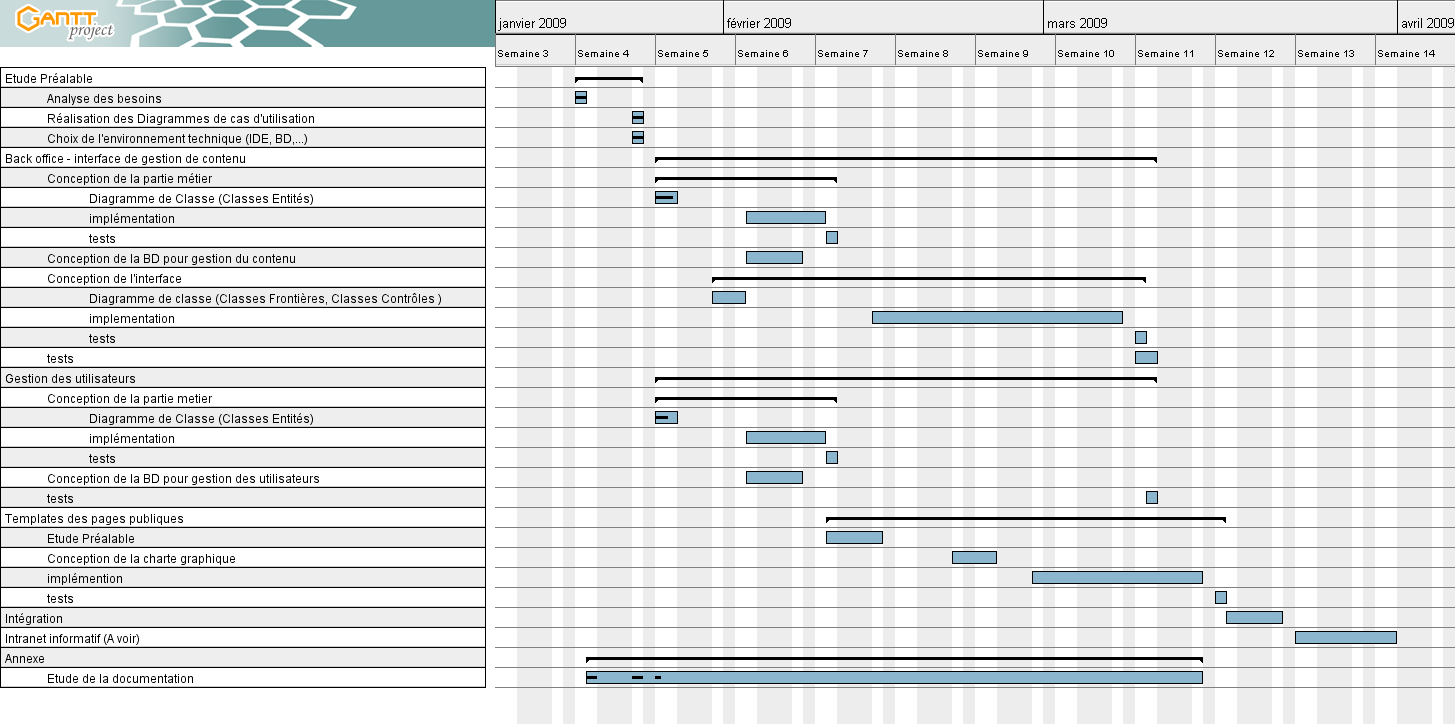


Figure : Planning prévisionnel

Tout d’abord, nous avons distingué cinq tâches importantes que sont l’étude préalable, la gestion du contenu, la gestion des droits utilisateurs, la réalisation des templates et l’intégration des parties Back End et Front-End. L’étude préalable : Cette première partie nous a permis d’étudier les besoins du client (l’entreprise IOcean) et par la suite de concevoir un diagramme de cas d’utilisation correspondant à une solution plausible au besoin. Enfin, nous avons déterminé les outils nécessaires à la réalisation du projet (voir environnement Technique). Gestion du Contenu et Gestion des droits des utilisateurs: Ces deux parties ont été réalisées avec la même démarche. Tout d’abord nous avons effectué la conception de la partie métier (diagramme de classes, implémentation et tests), ensuite la creation de la base de donnée et des tables nécessaires, et enfin nous avons dévellopé l’interfaces ( Vues JSF). Les Templates : Nous devions réaliser des templates pour permettre au client d’adapter notre réalisation en fonction de son application spécifique. Cette partie correspond à la création de composants Integration : Par intégration, nous regroupons les tâches suivantes : l’adaptation de la partie front-end à la partie back-end, la revue de code, l’ajout de commentaire, la correction de bug.

### Outil de collaboration et de versionning : SVN

Nous avons mis en place un Wiki avec Googlecode. En utilisant cette plateforme, nous avons pu partager des informations sur le projet ainsi que le code source grâce au SVN intégré de Google Code.

# Technologies utilisées et concepts

## Framework Seam

## Le modèle de composant Seam

A l'origine le modèle de composants de Seam a été conçu pour permettre l'intégration directe entre JSF et les EJB3.

Quand on développe une application web à partir de JSF, on implémente généralement deux types de composants :

* une page JSF qui va constituer la vue et contenir le code HTML,
* une classe Java appelée backing bean JSF qui va constituer le modèle et fournir les méthodes pour implémenter les actions utilisateurs (soumission formulaire ...).

Les méthodes du backing bean JSF liées aux actions utilisateurs accèdent en général à un composant métier. Si celui-ci est déployé sous la forme d'un EJB 3 de type Session Bean, il est alors nécessaire d'implémenter une couche d'adaptation entre ces deux types de composants.



Figure : Intégration JSF/EJB 3 sans le framework Seam

Avec Seam, il n'y a plus de distinction entre le composant backing bean et le composant métier ; ils forment un seul et unique composant qu'on appelle composant Seam. Une page JSF peut donc invoquer directement un composant métier déployé sous la forme d'un Session Bean.



Figure : Intégration JSF/EJB3 avec le framework Seam

Un composant Seam est une classe java du type POJO ou EJB 3.0 qui contient les annotations @name pour identifier le composant et @Scope pour le contexte applicatif.

Exemple : déclaration d'un composant Seam :

@Name("sessionUtilisateur")

@Scope(ScopeType.*SESSION*)

**public** **class** SessionUtilisateur {

…

}

Chaque composant Seam est déclaré sous un nom logique (ici sessionUtilisateur) et l'appel de ce composant depuis une page JSF s'effectue de la façon suivante :

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

Avec ce principe, les développements JSF sont simplifiés, puisque il n'y a plus besoin de déclarer les backing beans dans le fichier de configuration faces-config.xml.

La spécification Web Beans JSR 299 (http://jcp.org/en/jsr/detail?id=299) est basée sur les concepts de ce modèle de composant et sera très certainement intégrée dans la **spécification JEE6.**

## Contextes Seam

On accède à un composant Seam depuis son contexte. Chaque type de composant est déclaré par défaut dans un contexte mais il est facile d'étendre le contexte d'un composant. Pour cela, Seam met à disposition différents contextes :

* **stateless** qui est le contexte par défaut des composants EJB Stateless, sert pour les composants sans état.
* **request** (idem J2EE correspond à la requête HTTP) ou **event** qui est le contexte par défaut des composants du type POJO ou JavaBean,
* **page** (idem J2EE), le composant est accessible sur la durée de vie d'une page JSF,
* **conversation** qui correspond à un enchainement de pages.
* **session** (idem J2EE), le composant est accessible depuis la session HTTP,
* **process**, contexte utilisé pour la gestion des processus (par exemple JBPM)
* **application** (idem J2EE) est commun à toute l’application web et dure jusqu’à l’arret du serveur d’application.

## Inversion de contrôle et injection de dépendances

Pour que les composants puissent interagir entre eux, Seam propose un mécanisme d'injection des dépendances bijectif :

* un composant peut être injecté automatiquement dans un autre composant (équivalent au mécanisme d'injection des dépendances du framework Spring),
* un composant peut également créer et projeter un autre composant dans un contexte

(outjection) qui deviendra ainsi accessible par tous les autres composants.

Il est également possible de forcer la création ou non des composants au moment de l'injection.

Avec Seam, nous avons donc un mécanisme d'injection des dépendances complètement dynamique, ne nécessitant pas de configuration XML.

L’idée de l’inversion de contrôle en anglais Inversion Of Control (IoC) est de ne pas gérer manuellement dans chaque composant la récupération des autres composants dont il a besoin, mais de le laisser faire au framework. Les composants vont ainsi pouvoir être développés indépendamment les uns des autres, en laissant le soin au framework de faire le lien entre eux à l’exécution.

Chaque composant va signaler ses dépendances, et le framework va les lui injecter.

On indique une dépendance devant être injectée en ajoutant une annotation @In à une variable d’instance d’un composant.

Le framework va au-delà de l’injection de dépendance, puisqu’il va aussi gérer l’opération inverse, via l’annotation @Out, pour (re-)publier après l’exécution d’une méthode une dépendance si sa référence a été modifiée par la méthode.

## Gestion des conversations

Actuellement, pour gérer l'état conversationnel dans une application web, la seule solution est de stocker les informations au niveau de la session HTTP, ce qui entraîne souvent des problèmes de réinitialisation ou de purge des informations.

Avec Seam, on utilise ce qu'on appelle les conversations longues soit de manière implicite à partir de composants du type EJB3 stateful ou bien avec des JavaBeans en les déclarants dans le contexte de conversation.

Une conversation Seam va donc englober plusieurs requêtes et un utilisateur pourra utiliser plusieurs conversations sachant que chaque conversation est complètement indépendante.

Le démarrage d'une conversation longue s'effectue de manière explicite en positionnant par exemple une annotation Java @Begin au niveau d'une méthode métier.

De la même façon une conversation est terminée à partir de l'annotation @End sur une méthode ou bien dès qu'elle tombe en timeout.

## Intégration avec JPA

JPA (Java Persistence API) est l'API de persistance fournie par JEE5. Cette API se base en grande partie sur les concepts apportés par Hibernate, le framework incontournable de mapping relationnel objet (ORM).

JPA et Hibernate ont un mécanisme de chargement des données à la demande (Lazy Loading), pour éviter par exemple qu'un seul select sur une table parent récupère toutes les données des tables filles. Ce chargement des données à la demande n'est possible qu'à partir d'un contexte de persistance (ou session hibernate) et généralement celui-ci n'est plus accessible depuis une page JSF de l'application web, ce qui peut parfois poser des problèmes pour l'affichage des données.

Pour contourner ce problème, il y a deux solutions :

* implémenter une couche d'objets (DAO pour Data Acces Object) permettant de charger les objets de persistance avec toutes les données nécessaires pour l'affichage,
* utiliser le contournement OpenSessionInView qui consiste à maintenir le contexte de persistance ouvert pour chaque requête HTTP.

Ces deux solutions, si elles fonctionnent, ont leurs limites. Dans le cas des DAO, on duplique le code et, avec la deuxième solution, on maintient le contexte de persistance ouvert pour toutes les requêtes HTTP et ce mécanisme ne convient pas vraiment aux applications JSF.

Avec Seam, la solution est beaucoup plus simple : le contexte de persistance sera réutilisé si la page web a besoin d'afficher des données supplémentaires. On a donc plus besoin d'implémenter de **couche DAO.**

## Framework Hibernate

Hibernate, comme tous les autres outils de mapping objet/relationnel, nécessite des méta-données qui régissent la transformation des données d'une représentation vers l'autre (et vice versa). Dans Hibernate 2.x, les méta-données de mapping sont la plupart du temps déclarées dans des fichiers XML.

Maintenant, avec la spécification java EE5, les annotations sont disponibles pour arriver au même résultat avec une très grande élégance.

Eclipse, par exemple, prend en charge la complétion automatique et la coloration syntaxique des annotations du JDK 5.0. Les annotations sont compilées en bytecode et lues au moment de l'exécution (dans le cas d'Hibernate, au démarrage) en utilisant la réflexion, donc pas besoin de fichiers XML externes.

La spécification EJB3 reconnaît l'intérêt et le succès du paradigme du mapping objet/relationnel transparent. La spécification EJB3 standardise les APIs de base et les méta-données requises par n'importe quel mécanisme de persistance objet/relationnel. Hibernate EntityManager implémente les interfaces de programmation et les règles de cycle de vie telles que définies par la spécification de persistance EJB3. AvecHibernate Annotations, ce wrapper implémente une solution de persistance EJB3 complète (et autonome) au-dessus du noyau mature d'Hibernate.

On peut utiliser soit les trois ensembles, soit les annotations sans le cycle de vie et les interfaces de programmations EJB3, ou même Hibernate tout seul, selon les besoins techniques et fonctionnels du projet. On peut à tout moment recourir aux APIs natives d'Hibernate ou même, si besoin est, à celles de JDBC et au SQL.

Dans ce projet, nous avons utilisé Hibernate tout seul pour le mapping objet/relationnel car, le cycle de vie est géré par le framework Seam.

Conformément à la documentation officielle de Hibernate, nous avons créer un wraper connu sous le nom de HibernateUtil dans un contexte statique pour la gestion des transactions.

## Frameworks JSF, Facelets

# Configuration de l’environnement de développement

## Serveur d’application Tomcat 6.0

Un serveur d'applications est un serveur sur lequel sont installées les applications utilisées par les usagers (les clients). Ces applications sont déployées sur le serveur d'applications et les clients y accèdent à distance souvent par réseau. Dans l'entreprise actuelle, l'outil informatique (appelé aussi SI pour Système d'Information) a pris une place centrale et prépondérante. Il prend en charge des applications dans toutes les étapes du processus métier, de la production à la comptabilité. Ces applications, qui étaient souvent déployées sur différents postes (clients lourds), sont aujourd'hui de plus en plus centralisées sur des serveurs d'applications, sollicités par des postes clients plus nombreux et légers (clients légers, riches, ou semi-lourds). A l'inverse ces serveurs sont des systèmes de plus en plus lourds prenant en charge des applications de l'entreprise de plus en plus diverses et nombreuses. Un serveur d'applications est donc un serveur qui centralise toutes les applications utilisées par les postes clients. Les applications sont chargées sur le serveur tandis que leurs IHM (Interfaces Hommes-Machines) sont distribuées sur les postes clients. Dans une infrastructure N-tiers régulière, on peut déployer plusieurs serveurs d'applications, que ce soit pour répartir la charge lorsque le nombre élevé de postes clients est une exigence critique, ou que ce soit simplement pour les redonder lorsque leur disponibilité est aussi une exigence critique (les dispositifs de redondance peuvent être plus ou moins sophistiqués suivant qu'ils garantissent des temps de reprise en secours plus ou moins brefs, i.e. une disponibilité de service plus ou moins continue).

Le serveur d'applications agit comme tout serveur, il prend la requête du poste client, exécute les traitements à effectuer et retourne le résultat au poste client. Ce faisant, il assure la persistance des données au cours et entre plusieurs transactions d'un même poste client, ainsi que la persistance des données partagées et les arbitrages d'accès entre plusieurs postes clients concurrents.

Nous avons configuré le serveur d'application Tomcat 6.0 manuellement.

Pour ce faire, nous avons modifié le fichier server.xml et spécifié le contexte de notre application comme suit:

<Context docBase= "ter\_cms\_jboss\_seam" path="/ter"

reloadable="true" source="org.eclipse.jst.jee.server:ter\_cms\_jboss\_seam"/>

</Host>

L'élément Context doit être placé avant la fermeture de l'élément Host. Les attributs les plus importants sont les suivants :

• “docBase” permet de spécifier un nom de l'application sur le serveur d'application

• “path” correspond au chemin d'accès du contexte

• “reloadable” permet le redéploiement ou non de l'application web sur le serveur d'application

Ensuite nous avons inclus des fichiers .jar correspondant aux librairies nécessaires au fonctionnement de l'application dans le répertoire. Notamment le JDBC (java database connector) permettant de relier les objets persistants à la base de données.

Dans toutes applications web, tous les utilisateurs sont identifiés par une session. la session contient des informations sur l'utilisateur. Un exemple typique est celui du site de commerce : les produits du panier sont mémorisés pendant toute la session.

Dans le but d'éviter que les sessions s'accumulent très rapidement, il est nécessaire de les détruire après un certain temps d'inactivité de l'utilisateur. Pour cela, il faut définir la valeur du “session-timeout” dans le fichier web.xml de Tomcat:

<session-config>

<session-timeout>30</session-timeout>

</session-config>

## Hibernate

Parce qu'Hibernate a été conçut pour fonctionner avec des environnements différents, Il existe beaucoup de paramêtre de configuration. Nous n'allons donc pas détailler toute cette configuration mais simplement énoncé quelques une qui nous ont permi de configurer Hibernate pour l'application. Ces configurations se trouvent dans le fichier hibernate.cfg.xml.

Tout d'abord, nous devons spécifier les paramêtres de connection à la base de donnee. Dans l'extrait du fichier “hibernate.cfg.xml” ci-dessous, nous avons décrit les propriétés nécessaires permettant de se connecter à notre base de donnée MySql en indiquant, notamment, la propriété de génération automatique de la base de donnée “hbm2ddl.auto”. En effet, ceci permet, grâce aux annotations de Java Persistance API mentionnées dans les classe Java de créer les tables dans la base de donnée.

<property name="connection.url">jdbc:mysql://localhost/cms</property>

<property name="connection.username">root</property>

<property name="connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="connection.password"></property>

<property name="transaction.factory\_class">

org.hibernate.transaction.JDBCTransactionFactory</property>

<property name="hbm2ddl.auto" >update</property>

Ces classes Java doivent, cependant, être mentionnée dans ce même fichier de configuration :

<mapping class="entite.File"/>

<mapping class="entite.Contenu"/>

<mapping class="entite.ContenuText"/>

<mapping class="entite.Rubrique"/>

<mapping class="entite.Article"/>

<mapping class="entite.Nouvelle"/>

<mapping class="entite.Utilisateur"/>

# Mise en œuvre et réalisation

## Diagrammes des cas d’utilisation

Le travail consiste à concevoir une plateforme de CMS permettant à différents utilisateurs de participer de façon collaboratif à la publication d’un contenu avec de rôles différents. Pour réaliser ce TER nous nous somme basé sur les exigences du cahier de charges fournis par IOcean. En effet Ce travail a été réalisé en collaboration avec l’entreprise IOcean qui nous a fourni les exigences du cahier de charge en tant que client. La plate forme du CMS se divise en deux grandes parties, une partie publique et une partie privée communément nommées front end et back end. Le front end est la partie de la plate forme qui interagit directement avec l’utilisateur. Le back end est la partie cachée à l’utilisateur où s’effectuent le traitement de données et la gestion des droits des utilisateurs. Dans chacune des ces deux parties nous avons différent utilisateurs qui jouent des rôles différents d’une partie à l’autre. L’analyse de ces exigences nous a permis dans un premier temps d’élaborer les diagrammes de cas d’utilisation de la figure 1 et 2 pour la partie back end et front end. Ce travail nous a permis de bien appréhender les difficultés de la gestion du contenu et des droits des utilisateurs sur ce contenu. Ainsi par le biais de l’interface d’administration dans la partie back end, la plate forme du CMS offre la possibilité de gérer les utilisateurs et leurs droits. C’est à dire de leur donner accès ou non aux fonctionnalités du système. Dans chacune des parties (back end et front end ) nous avons des fonctionnalités multiples et des utilisateurs qui jouent des rôles différents d’une partie à l’autre.

Dans la partie front end nous avons deux acteurs, un visiteur et un membre , le visiteur est un utilisateur quelconque qui a la possibilité de consulter les contenus public et s’inscrire si il le souhaite pour avoir certain droits sur les informations(contenus) réservés au membre et le membre a en plus la possibilité de consulter les contenus réservés aux membre une fois qu’il s’est authentifié.

Dans la partie back end nous avons trois différents acteurs, un rédacteur, un gestionnaire et un administrateur. Chacun des ces acteurs à un rôle qui lui est spécifique et qui lui donne les droits d’accès ou non à certaines fonctionnalités du partie back end. Ainsi pour éditer, publier, modifier ou supprimer des articles ou des rubriques sur cette plate forme les différents acteurs doivent avoir les permissions appropriées. Seuls les utilisateurs qui ont un droit d’accès back end pourront automatiquement accéder à l’interface d’administration (au back end) du site.

Le rédacteur a le droit d’ajouter un contenu (éditer) et de le modifier à sa guise mais sans droit de publication.

Le gestionnaire a le droit du rédacteur et il a aussi le droit de publier et dépublier un contenu et de modifier un article mais pour accéder à ces fonctionnalités il doit tout d’abord s’identifier en tant que gestionnaire. En effet c’est une fois que le gestionnaire est bien identifié qu’il est accès aux contenus aux quels il le droit de gérer. L’administrateur, il a accès à tous les fonctionnalités de la partie back end. C’est l’utilisateur par défaut qui est créé lors de l’installation de la plate forme et qui gère tous les utilisateurs qui on accès à un contenu.

Lorsqu’un gestionnaire est accès à la partie back end, il ne voit que les fonctionnalités dont il a le droit de gérer. Son espace d’administration est ainsi restreint par rapport à celui de l’administrateur qui se voit offert tous les fonctionnalités du système. Dans le cas où un utilisateur tente de réaliser ou effectuer une action dont il n’a pas le droit, une exception est levé lui indiquant qui n’a pas le droit d’accès à cette fonctionnalité.

Les droits de chaque utilisateur sont gérés par rapport à un contenu. C’est à dire que chaque contenu a des droits sur l’utilisateur. Par exemple le contenu d’un rédacteur a le droit d’être modifié par son propre rédacteur et a le droit d’être publier ou d’être retiré de la publication par le gestionnaire de ce contenu ou du rubrique de ce contenu. Ce même contenu a le droit d’être supprimé ou mis à la corbeille par l’administrateur. Nous avons donc géré l’implémentation de telle façon que chaque contenu connais ses rédacteurs et ses gestionnaires.

Nous avons ainsi réalisé une classe de test nous permettant de tester tout ce qu’un utilisateur avec un rôle qui lui est spécifique peut faire ou ne pas faire sur un contenu. Nous vérifions aussi dans ce classe de test sur l’état d’un contenu, qui peut être publié, non publié ou mis en corbeille et qui a le droit de modifier l’état d’un contenu.

### Back end

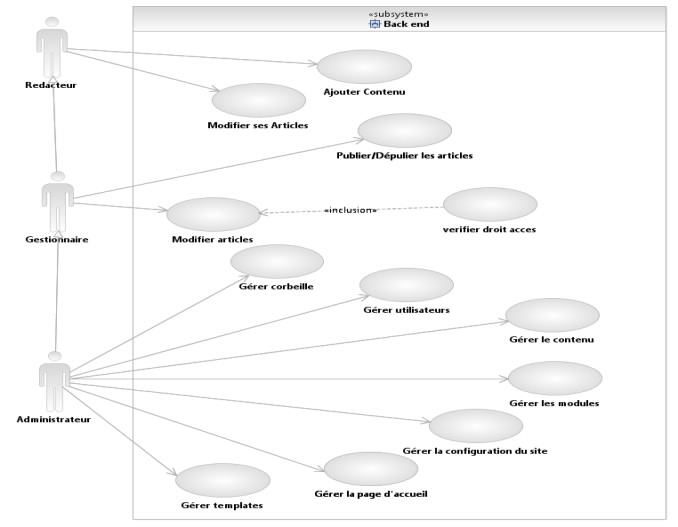


Figure : Diagramme de cas d'utilisation Back-end

### Front end

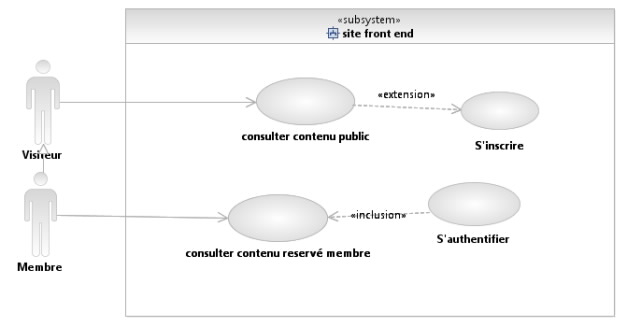


Figure : Diagramme de cas d'utilisation Front-end

## Gestion du contenu

### Structure de données

### Design pattern composite

## Gestion des droits d’utilisateurs

### contrainte sur l'implémentation des utilisateurs

D'après le cahier des charges, le C.M.S. doit pouvoir s'adapter à une application spécifique. Ceci implique que, dans le cas où cette application implémente une gestion des utilisateurs, le C.M.S. devra s'en servir pour ne pas avoir à gérer les droits de deux façons séparées. De plus, cette adaptation devra s'effectuer le plus simplement possible avec le minimum de changement dans le code dans le but de faciliter l'intégration du C.M.S.

### solution adoptée

La solution retenu est de créer une interface implémentable appelée IUtilisateur (voir figure 1.1.2). Il suffit d'implémenter cette interface avec une nouvelle classe ou avec une classe déjà existante dans l'application spécifique. Chaque appel à une propriété d'un utilisateur dans le C.M.S. fera appel à cette interface ce qui permet une adaptation simplifiée à l'application spécifique.

Figure 1.1.2: interface IUtilisateur.java

### modification à effectuer pour l'adaptation

Malgré l'utilisation d'une interface, certains changements doivent être tout de même appliqués dans les fichier suivants :

fichier de configuration d'Hibernate - Le mapping de la classe Utilisateur est necessaire au bon fonctionnement de l'application. Cette classe concrête doit implémenter la classe IUtilisateur soit par une classe de l'application spécifique, soit par une nouvelle classe. Le chemin d'accès à cette classe doit donc être modifié selon le cas de figure.

Figure 1.1.3: hibernate.cfg.xml

La classe Contenu et la classe Rubrique - De la même manière que précedemment, une modification peut-être necessaire pour faire le mapping des objets persistents Rubrique et Contenu.

Figure 1.1.4: classe Contenu.java

Figure 1.1.5: classe Rubrique.java

Le composant d'initialisation du C.M.S. doit également être modifier. L'attribut admin doit avoir le même type que celui de la classe implémentant l'interface IUtilisateur.

Figure 1.1.6: composant Initialisation.java

## Gestion des droits

### implémentation des droits

Comme nous l'avons vu précédemment, chaque contenu est classé selon une hiérarchie arborescente (voir Figure X). Cette classification va nous permettre de gérer les droits des utilisateurs sur le contenu. Nous avons défini trois statuts d'utilisateurs ayant accès à la partie d'administration du C.M.S. (voir Figure X : diag de use case Back End). Selon leur statut, ces utilisateurs auront des droits différents.

Le statut administrateur est géré très simplement avec une propriété directement liée à l'utilisateur qui indique si ce dernier est administrateur ou non (Figure 2.1.1 ).

Figure 2.1.1: extrait de IUtilisateur.java

La gestion des statuts gestionnaire et rédacteur est plus délicate. En effet, selon les spécifications du cahier des charges, un utilisateur pourra être gestionnaire d’une rubrique, seulement rédacteur pour une autre rubrique, et n’avoir aucun droit sur une troisième rubrique. Pour répondre à cette contrainte, la solution adoptée a été d'ajouter à chaque rubrique deux listes d'utilisateurs (Voir Figure 2.1.2). Une première liste contient les gestionnaires de la rubrique et une seconde liste contient les rédacteurs de la rubrique. Ainsi, chaque rubrique connaîtra, par l'intermédiaire de ses listes, si un utilisateur à des droits

Figure 2.1.2: extrait de Rubrique.java

### comment gérer les droits ?

L'affectation du statut administrateur s'effectue via l'interface de Modification d'un utilisateur (voir Figure 2.2.1).

Figure 2.2.1: sreenshot - Modification d'un Utilisateur par un administrateur

L'affectation des statuts gestionnaire et rédacteur dépendent des rubriques . Pour chaque rubrique, un utilisateur ayant les droits nécessaire (voir section suivante) peut définir les statut via le lien 'Definir Statut' (voir Figure 2.2.1). Ensuite l'utilisateur aura la liste des utilisateurs inscrit et pourra ajouter, modifier ou supprimer des droits à sa convenance (voir Figure 2.2.2).

Figure 2.2.1: screenshot - Gestion des Rubriques par un administrateur

Figure 2.2.2: screenshot - Choix des Gestionnaires et Rédacteur

### Qui affecte les droits aux utilisateurs ?

Le statut administrateur peut seulement être modifier par un utilisateur administrateur. A l'initialisation du C.M.S., un administrateur est automatiquement créé, ce dernier pourra, par conséquent, affecter de nouveaux utilisateur avec le statut d'administrateur (voir Figure 2.2.1).

Le statut gestionnaire peut être modifier par un utilisateur administrateur ou par un utilisateur gestionnaire de la rubrique parent de la rubrique à modifier (voir Annexe : Classe GestionContenu, méthode droitDefinirGestionnaire() ).

Le statut rédacteur peut être modifier par un utilisateur administrateur ou par un utilisateur gestionnaire de la rubrique à modifier (voir Annexe : Classe GestionContenu, méthode droitDefinirRedacteur() ).

### Conséquence des droits sur le contenu

Exemple : L'utilisateur “soule” est affecté en tant que rédacteur de la rubrique “faculté des sciences um2”

• après s'être authentifier, “soule” peut ajouter un contenu à n'importe quel niveau de la hierarchie descendante de la rubrique “faculté des sciences um2”

• “soule” peut modifier les contenus qu'il a créé mais n'a pas le droit de publication et de suppression

## Templates de pages web

Le templating consiste à factoriser la structure commune d'un ensemble de pages et à l'extraire dans une nouvelle page, appelée template ou modèle. Les autres pages utilisent alors le template comme structure et y injectent leur contenu spécifique.

Le templating offre plusieurs avantages :

* Uniformiser la structure des pages.
* Simplifier la mise à jour : une modification dans le template se propage automatiquement dans toutes les pages qui l'utilisent.
* Gain en productivité : moins de code à écrire : une page ne contient que ce qui lui est propre.

Grace à l’utilisation des facelets dans notre application, nous avons pu faire du templating, qui est non seulement utile pour éviter la recopie de code mais aussi pour la réutilisation des composants. Les deux tempates se trouve dans le répertoire /template.

Chaque page de l’application (.xhtml) est donc une composition d’une page template-admin.xhtml, qui affiche l’entête de la page, le menu de navigation, le pied de page, dont on redéfinie le contenu. Dans le cas de la partie « Backend » chaque page est une composition de template-admin.xhtml qui est une composition de header-admin.xhtml en y ajoutant le menu.xhmtl et footer.xhtml. La page admin-xhtml utilise ce template et rédéfinit le contenu afin d’acquérir l’élément courant grâce à un composant seam qui retourne la page courante à insérer.

Le squelette du template de la partie administration est le suivant :

<body>

<ui:insert name=*"header"*>

<ui:include src=*"header.xhtml"*/>

</ui:insert>

<ui:insert name=*"navigation"*>

<ui:include src=*"navigation.xhtml"*/>

</ui:insert>

<ui:insert name=*"content"* />

<ui:insert name=*"footer"*>

<ui:include src=*"footer.xhtml"*/>

</ui:insert>

</body>

Les balises importantes qu’il convient d’expliquer sont :

<ui :include> : Permet

<ui :insert> : Permet

<ui :composition> : Permet

Nous remarquons que seule la partie <ui:insert name=*"content"* /> sera redéfinit par la page cliente utilisant ce template, ce qui rend cette partie du template dynamique et réutilisable.

On commence par spécifier que l'on utilise un template avec la balise <ui:composition> qui prend comme paramètre le chemin vers le fichier contenant le template. Pour définir les différents blocs qui seront injectés dans le template, on utilise la balise <ui:define>. Cette balise prend comme paramètre le nom logique du bloc correspondant dans le template et contient dans son corps le contenu à injecter dans le template.

Ainsi la page cliente *admin.xhtml* définira la balise :

<ui:composition template=*"template/template-admin.xhtml"*>

</ui:composition>

## Gestion des exceptions

Seam nous permet d’intercepter toutes les exceptions qui se produisent en dehors du contexte applicatif d’un composant de deux manières qui peuvent être utilisées conjointement :

* Par annotation des classes d’exceptions, Exemple :

*@Redirect(viewId="/error.xhtml", message="erreur contenu")*

*public class ContenuException extends RuntimeException { ... }*

Cette exception se traduit par un message à l’utilisateur et l’annulation de la transaction courante avec une redirection vers la page *error.xhtml.*

* Par le fichier de configuration *pages.xml :* Cette méthode a l’avantage de pouvoir intercepter toute sorte d’exceptions.

Ces deux possibilités sont complémentaires car, on ne peut annoter toutes les exceptions qu’on veut, c’est pourquoi Seam autorise l’utilisation du fichier XML afin d’intercepter ces dernières.

Pour que Seam soit capable d’intercepter des exceptions, il faut tout d’abord déclaré le master servlet dans le fichier web.xml de la manière suivante :

*<filter>*

*<filter-name>Seam Filter</filter-name>*

*<filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>*

*</filter>*

*<filter-mapping>*

*<filter-name>Seam Filter</filter-name>*

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

*</filter-mapping>*

Dans ce projet, nous avons choisi la deuxième méthode celle à l’aide du fichier XML car, elle permet d’intercepter toute sorte d’exception se produisant en dehors du contexte applicatif d’un composant Seam.

Par exemple pour intercepter une exception et afficher un message compréhensible à l’utilisateur quand sa session a expiré :

# Conclusion

# Glossaire

# Bibliographie

# Annexes