**2011 - 2012**



**IUT de Lens**

*BAVIER Romain*

*HANOCQ Alexandre*

*MOOTINAL Gerardo*



**NOTE DE SYNTHÈSE TECHNIQUE**

**DE PROJET**

Création d’une application pour le suivi des étudiants de l’IUT

Sommaire

[1. Introduction 3](#_Toc317341405)

[2. Cahier des charges 3](#_Toc317341406)

[2.1. Objectifs 3](#_Toc317341407)

[2.2. Contraintes technologiques 3](#_Toc317341408)

[2.3 Analyse des Besoins et MCD 3](#_Toc317341409)

[3. Partie Technique 5](#_Toc317341410)

[3.1 Mode de fonctionnement. 5](#_Toc317341411)

[3.2 Le MVC 6](#_Toc317341412)

[3.2.1. Le Modèle 7](#_Toc317341413)

[3.2.1.1. Les Entités 7](#_Toc317341414)

[3.3.1.2. L’accès aux données 9](#_Toc317341415)

[3.2.2 Le contrôleur 11](#_Toc317341416)

[3.2.3 Les vues 13](#_Toc317341417)

[3.3 Ajout de fonctionnalité 15](#_Toc317341418)

[3.3.1 Génération de PDF avec la librairie itext. 15](#_Toc317341419)

[3.3.2 Génération de document EXCEL avec la librairie POI. 17](#_Toc317341420)

[3.4 Sécurité 17](#_Toc317341421)

[3.4.1 Cryptage des mots de passe. 17](#_Toc317341422)

[3.4.2 Sécurité des pages. 18](#_Toc317341423)

[3.4.3 Test unitaire. 18](#_Toc317341424)

[4. Conclusion 18](#_Toc317341425)

# Introduction

Afin de mettre en pratique nos connaissances, et les notions enseignées en licence professionnelle Système Informatiques et Logiciels option Sécurité Informatique à l’IUT de Lens, nous, Romain BAVIER, Alexandre Hanock et Gerardo Mootinal devions réaliser un projet tuteuré informatique d’une durée de six semaines ceci afin de valider notre second semestre. Ce projet s’est déroulé du 02/01/2012 au 17/03/2012.

# Cahier des charges

## 2.1. Objectifs

L’objectif du projet était de développer une application permettant d’avoir un suivi scolaire et professionnel de chaque étudiant qu’il soit ancien ou nouvel étudiant de l’IUT. Les personnes habilitées à utiliser l’application seront des personnes internes à l’IUT tels que les enseignants, le personnel administratif. Ils pourront ainsi suivant leur droit, visualiser, modifier ou ajouter des informations concernant les étudiants ainsi que sur leur suivi.

## 2.2. Contraintes technologiques

Les technologies utilisées pour ce projet ont été imposées dans le sujet.

Ces technologies étaient J2EE6 associé au Framework JSF 2.0 (Kit de composants logiciels structurels, qui servent à créer les fondations ainsi que les grandes lignes de tout ou d’une partie d'un logiciel, c'est-à-dire l’architecture du projet).

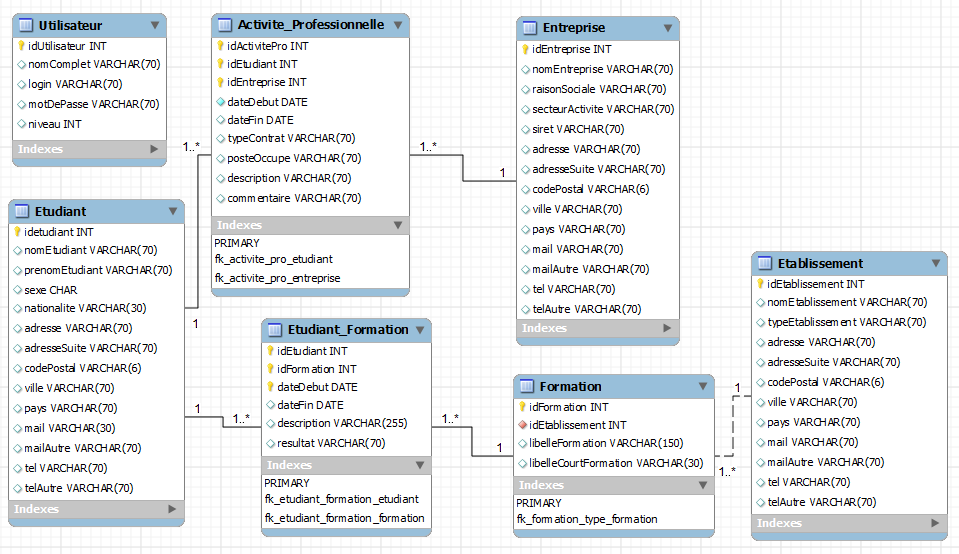
Pour ce qui est de la persistance de nos données, nous avons choisi eclipseLink, Framework open source de mapping objet-relationnel, fournissant une plateforme puissante et flexible permettant de stocker des objets Java dans une base de données relationnelles. Il est un dérivé de TopLink de la société Oracle.

En ce qui concerne la base de données, nous avons utilisé dans un premier temps derby pour des raisons de simplicité, et dans un second temps nous sommes passés sur le SGSBD POSTGREY.

Enfin le serveur d’application utilisé était glassfish.

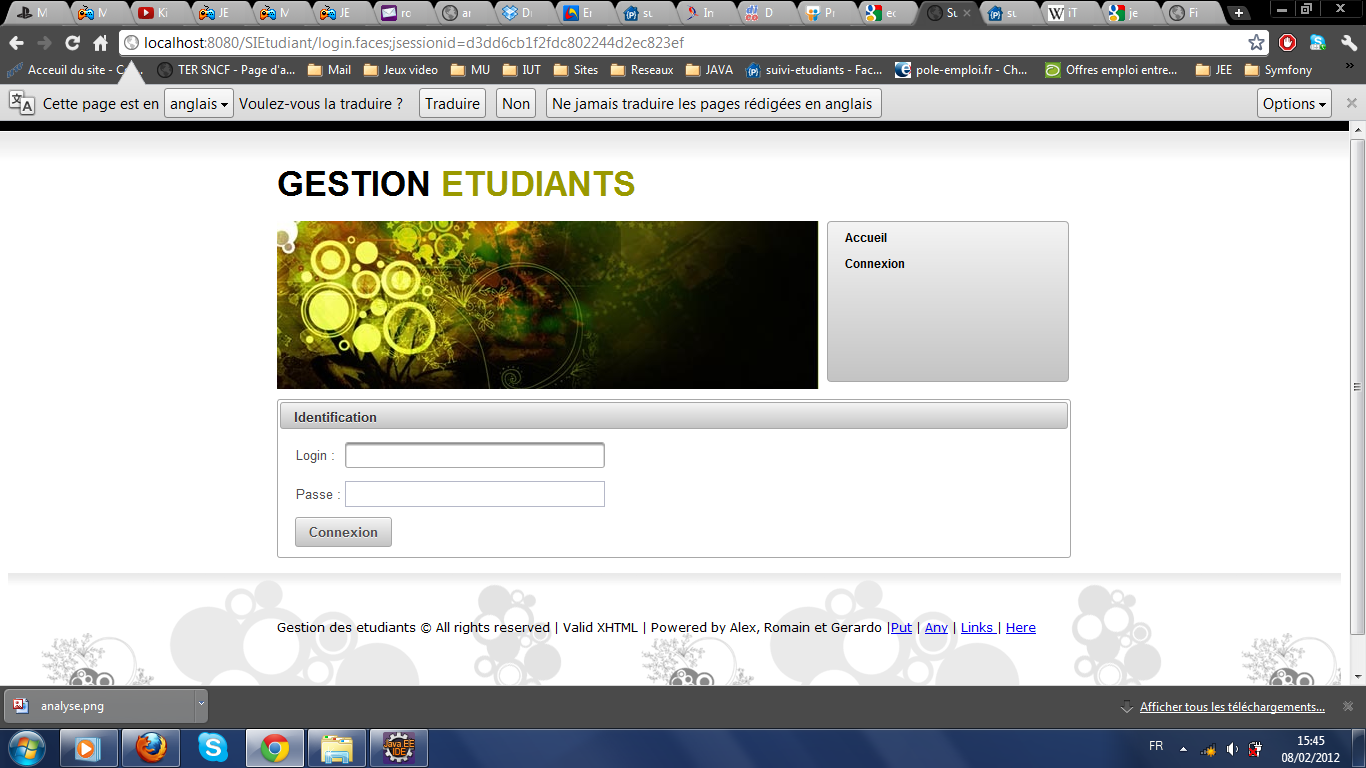
## 2.3 Analyse des Besoins et MCD

Nous avons dans un premier temps réalisé le MCD (modèle conceptuel de données), que nous avons complété/modifié en fonction des ajouts de fonctionnalités imposées par notre tuteur de projet, Mr Hémery. Tout cela, pour répondre au mieux aux attentes du sujet.



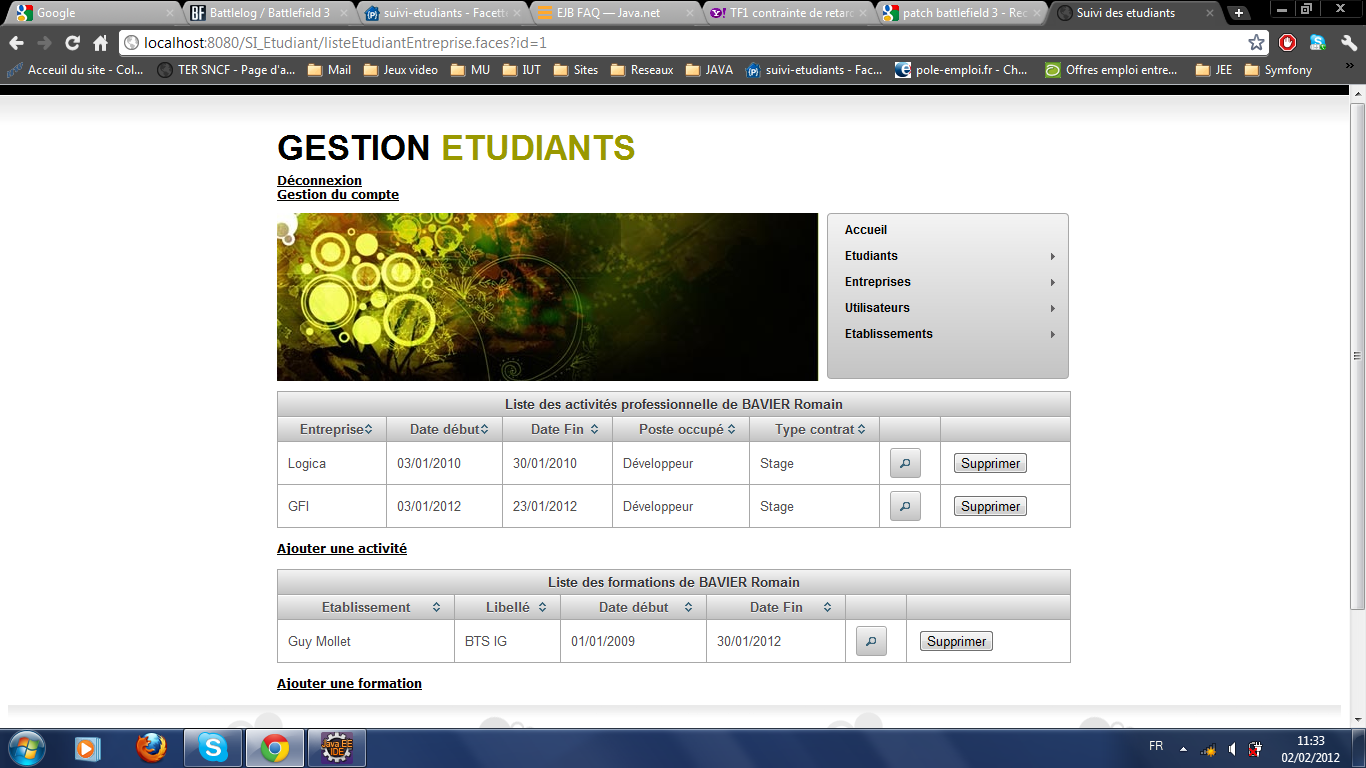
# Partie Technique

## 3.1 Mode de fonctionnement.



*Page de login (Non connecté)*

Ci-dessus, la page d’accueil de l’application. Cette page permet à l’aide d’un identifiant et d’un mot de passe, de se connecter à l’application.



*Page activité d'un étudiant (connecté)*

Une fois la connexion établie, l’utilisateur accèdera à l’espace de gestion. Il pourra, par exemple visualiser la liste des activités et formations de l’étudiant qu’il aura sélectionné au préalable.

En fonction du rôle de l’utilisateur, le menu à droite affiche ou non certains sous menu. Ce menu permet la navigation entre les différentes pages et permet d’accéder aux fonctions de l’application et ainsi pouvoir consulter, ajouter, modifier les données.

# 

## 3.2 Le MVC

Notre application utilise le Framework JSF. Ce Framework est basé sur le design pattern Modèle-Vue-Contrôleur (MVC). Ce pattern architectural sépare les données (le modèle), l'interface homme-machine (la vue) et la logique de contrôle (le contrôleur). Ce point nous parait important car c’est de cette manière que notre code est organisé. Bien entendu cette partie fut créée après avoir écrit nos tables sql à partir de notre MCD.

### 3.2.1. Le Modèle

Cette partie représente  les données de notre application. Elle définit aussi l'interaction avec la base de données et le traitement de ces données.

#### 3.2.1.1. Les Entités

Afin que l’api de persistence décrit plus haut (JPA) puisse faire le lien entre un objet Java et la base de données, les classes à mapper sont définies en tant qu’entité JPA.

Une entité permet d'encapsuler les données d'une occurrence d'une table.

Elle est composée de propriétés qui seront mappées sur les champs de la table concernée. Chaque propriété encapsule les données d'un champ d'une table et sont utilisables au travers de simple accesseurs (getter/setter).

Ainsi chaque table de notre base de données possède son entité. Les entités de notre code sont plus ou moins identiques.

Ci-dessous, un extrait de l’entité Etudiant.

@Entity

@Table(name = "ETUDIANT")

**public** **class** Etudiant **implements** Serializable {

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** **long** id;

**private** String nom;

**private** String prenom;

**private** Adresse adresse = **new** Adresse();

**private** Contact contact = **new** Contact();

**private** Set<EtudiantFormation> lesFormations = **new** HashSet<EtudiantFormation>();

**private** Set<EtudiantEntreprise> lesEntreprises = **new** HashSet<EtudiantEntreprise>();

**public** Etudiant() {

**super**();

}

@OneToMany(mappedBy = "etudiant", cascade = CascadeType.*ALL*, fetch = FetchType.*LAZY*, orphanRemoval = **true**)

**public** Set<EtudiantEntreprise> getLesEntreprises() {

**return** lesEntreprises;

}

**public** **void** setLesEntreprises(Set<EtudiantEntreprise> lesEntreprises) {

**this**.lesEntreprises = lesEntreprises;

}

@OneToMany(mappedBy = "etudiant", cascade = CascadeType.*ALL*, fetch = FetchType.*LAZY*, orphanRemoval = **true**)

**public** Set<EtudiantFormation> getLesFormations() {

**return** lesFormations;

}

**public** **void** setLesFormations(Set<EtudiantFormation> lesFormations) {

**this**.lesFormations = lesFormations;

}

@Id

@GeneratedValue

@Column(name = "ETUDIANT\_ID")

**public** **long** getId() {

**return** id;

}

**public** **void** setId(**long** id) {

**this**.id = id;

}

…

}

A noter que les annotations présentes dans cette classe dépendent de l’API de persistance utilisé, ici eclipseLink.

Les annotations de type oneToMany (un à plusieurs) permettent de récupérer directement l’objet concerné.

@Table permet de faire la correspondance avec une table de la base de données.

@OneToMany(mappedBy = "etudiant", cascade = CascadeType.*ALL*, fetch = FetchType.*LAZY*, orphanRemoval = **true**).

Cette ligne va permettre de relier les formations de l’étudiant à la classe etudiantFormation.java.

cascade = CascadeType.*ALL* supprimera automatiquement les formations et entreprises de l’étudiant lors de la suppression de l’étudiant. En absence de cette ligne, la contrainte d’intégrité référentielle ne sera pas respectée et générera une erreur.

#### 

#### 3.3.1.2. L’accès aux données

C’est ici que l’application accèdera aux données de la base de données.

Chaque table de notre base de données, comme pour l’entité, possède une classe qui permettra d’effectuer des requêtes interagissant avec la base de données. Ces requêtes auront pour but de récupérer les éléments de la table dans l’application (select), ajouter, modifier et supprimer des informations sur les tables souhaitées.

@LocalBean

@Stateless

**public** **class** EtudiantEJB **implements** EtudiantEJBInterface{

@PersistenceContext(unitName = "si\_etu")

**private** EntityManager em;

/\*

\* Recupere tous les etudiants

\*/

@SuppressWarnings("unchecked")

**public** List<Etudiant> findAllEtudiants() **throws** EJBException {

List<Etudiant> results = em.createQuery("select e from Etudiant e")

.getResultList();

**return** results;

}

/\*

\* Cree un etudiant dans la BDD

\*/

**public** Etudiant createEtudiant(Etudiant etudiant) **throws** EJBException {

em.persist(etudiant);

**return** etudiant;

}

/\*

\* Supprime un etudiant

\*/

**public** **void** removeEtudiant(Etudiant etudiant) **throws** EJBException {

// em.merge(etudiant) => entité doit être détaché du bean sinon cela ne

// fonctionne pas

em.remove(em.merge(etudiant));

}

/\*

\* Met a jour un etudiant

\*/

**public** **void** updateEtudiant(Etudiant etudiant) {

em.merge(etudiant);

}

…

}

Ici, nous avons utilisé un EJB dit session dans nos classes.

En effet, un EJB (Enterprise JavaBeans) est une architecture de composants logiciels côté serveur pour la plateforme de développement J2EE6.

Cette architecture propose un cadre pour créer des composants distribués et hébergés au sein d’un serveur applicatif permettant, ici de représenter des services avec ou sans conservation d’état (@Statefull ou @Stateless)(EJB Session).

Au niveau de la requête select, le langage n’est pas du SQL mais du JPQL (Java Persistence Query Language). Dans ce type de requête, on ne manipule pas la table directement mais les classes entités écrites précédemment.  Il offre une approche objet et ses similitudes avec le langage SQL garantissent une facilité d'utilisation pour toutes personnes ayant déjà pratiqué le SQL.

En résumé,

[PersistenceContext](http://java.sun.com/javaee/5/docs/api/javax/persistence/PersistenceContext.html)(unitName = « si\_etu »), permet à l'EJB d'injecter le contexte de persistance si\_etu et de récupérer une instance de l'interface [EntityManager](http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html). Celle-ci est l’objet qui va permettre de manipuler les objets *Java* devenus des entités et ainsi lui permettre de les persister, comme

- l'ajout

- la lecture ;

- la mise à jour ;

- la suppression.

Cette instance est préparée par le serveur d'application à partir du fichier persistence.xml.

Voici donc la configuration de notre contexte de persistence.xml si\_etu contenu dans le répertoire META-INF, répertoire contenant les métas donnés, c'est-à-dire les fichiers descripteurs de déploiement de l’application.

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

<persistence version=*"1.0"*

xmlns=*"http://java.sun.com/xml/ns/persistence"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"*

*http://java.sun.com/xml/ns/persistence*

*http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd"*>

<persistence-unit name=*"si\_etu"* transaction-type=*"JTA"*>

<!-- <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider> -->

<jta-data-source>jdbc/gestionEtu</jta-data-source>

<class>entity.Utilisateur</class>

<class>entity.Etudiant</class>

<class>entity.Entreprise</class>

<class>entity.EtudiantEntreprise</class>

<class>entity.EtudiantFormation</class>

<class>entity.EtudiantFormationPK</class>

<class>entity.EtudiantEntreprisePK</class>

<class>entity.Formation</class>

<class>entity.Contact</class>

<class>entity.Adresse</class>

<class>entity.TypeFormation</class>

<properties>

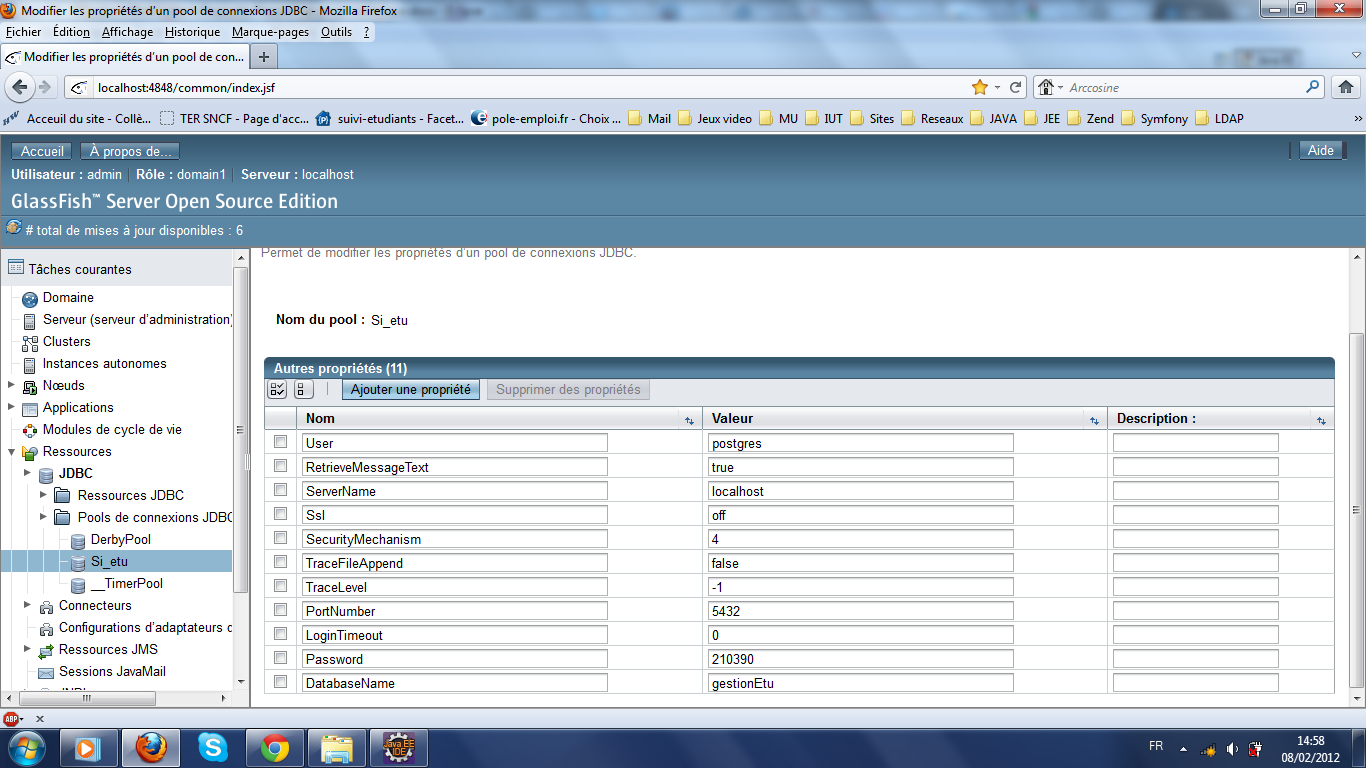
<property name=*"eclipselink.logging.level"* value=*"FINE"* />

</properties>

</persistence-unit>

</persistence>

La ligne <jta-data-source>jdbc/gestionEtu</jta-data-source> permet de connecter le contexte de persistance à un pool de connexion se trouvant sur le serveur d’application glassfish, lui-même connecté à la base de données. Il nous a fallu donc configurer tout cela dans la console d’administration du serveur d’application glassfish.



### 3.2.2 Le contrôleur

Le contrôleur est en quelque sorte une passerelle entre l’utilisateur et les données.

C’est ici que l’on définit les fonctions permettant la navigation entre les pages.

Voici un exemple simplifié de code.

@Named(value = "etudiantBean")

@ConversationScoped

**public** **class** EtudiantBean **implements** Serializable {

**private** **static** **final** **long** *serialVersionUID* = 1L;

@Inject

Conversation conversation;

@EJB

**private** EtudiantEJB etudiantEJB;

**private** Etudiant etudiant = **new** Etudiant();

**private** List<Etudiant> etudiants = **new** ArrayList<Etudiant>();

**private** Etudiant editEtudiant;

**private** Etudiant selectedEtudiant;

@PostConstruct

**public** **void** init() {

**if** (conversation.isTransient()) {

conversation.begin();

}

**try** {

etudiants = etudiantEJB.findAllEtudiants();

}

} **catch** (EJBException e) {

Redirection.*erreurXhtml*();

}

}

**public** String ajout() {

// Ajout

**this**.etudiantEJB.createEtudiant(etudiant);

// Fin

conversation.end();

Redirection.*listeEtudiants*();

**return** "listEtudiant";

}

// procédure pemettant de supprimer les etudiants selectionnés de la liste

// checked

**public** **void** supprimer() {

**for** (Etudiant unEtudiant : etudiants) {

**if** (checked.get(unEtudiant.getId())) {

etudiantEJB.removeEtudiant(unEtudiant);

}

}

conversation.end();

Redirection.*listeEtudiants*();

}

// fonction permettant de modifier un etudiant

**public** String modifier() {

etudiantEJB.updateEtudiant(editEtudiant);

conversation.end();

**return** "listEtudiant";

}

}

En JSF, c’est ce que l’on appelle BackingBean qui fait office de contrôleur.

Nous n’avons pas configuré les beans dans le fichier facesConfig.xml, mais choisi de mettre des annotations directement dans les fichiers JAVA concernés, pour des raisons d’ergonomie afin de rendre le code le plus lisible que possible.

En ce qui concerne les scopes du framework JSF, nous avons utilisé pour pratiquement tous les beans des conversationScoped et viewscoped.

En effet, ceux-ci permettent de garder plus longtemps en mémoire un bean ainsi que les valeurs inscrites. Il est donc possible d’ouvrir une conversation dans une page, d’y instancier des objets, de les récupérer dans une autre page, et de fermer la conversation.

Un autre avantage, est que les conversationscoped sont compatibles avec ajax.

Pour effectuer cela, on ajoute @ConversationScoped dans le backing bean et on injecte une variable de type Conversation. On appelle ensuite conversation.begin() et conversation.end() pour contrôler la création et la suppression de l'instance du bean.

L’annotation Postconstruct quant à elle, est utilisée sur une méthode qui doit être exécutée automatiquement après l’appel d’un bean.

Ici la méthode initialise directement nos listes et nous évite d’appeler explicitement la méthode init() dans nos vues. On peut dire que cette méthode fait un peu office de « construteur ».

De plus c’est ici que l’on va appeler les fonctions des EJB pour ajouter, modifier et supprimer. Ces fonctions seront déclenchées à travers d’évènements dans la vue.

### 3.2.3 Les vues

Elles correspondent à l’affichage de notre site. C’est ici que se trouvent les différents **formulaires** et **listes** que l’utilisateur voit à l’écran.

Nous avons utilisé dans un premier temps les balises *Facelets,* technologie de présentation pour le développement d'applications web en *Java* propre au Framework JSF 2.0 (langage EL) et dans un second les balises primefaces.

*Facelets* est en fait une page *JSP* transformée en une *Servlet* qui possède un cycle de vie différent de celui de *JSF*, ce qui peut être source de confusion et de problèmes. A l'inverse, *Facelets* est spécifiquement développé pour *JSF* et est plus performant et léger.

Cette technologieest basée sur *xml*, c'est pour cette raison que les vues sous *facelets* sont des pages *xhtml* et qu'elles doivent impérativement respecter la structure d'un document *xml*.

L’appel des fonctions définit précédemment dans le contrôleur s’effectue de la manière suivante. Ici nous affichons un tableau d’étudiants.

<p:dataTable id=*"tabEtu"* var=*"et"* value=*"#{etudiantBean.etudiants}"*

paginator=*"true"* rows=*"10"*>

<f:facet name=*"header"*>

Liste des étudiants

<p:column sortBy=*"#{et.nom}"*>

<f:facet name=*"header"*>

<h:outputText value=*"Nom"* />

</f:facet>

<h:outputLink id=*"lien1"* value=*"listeEtudiantEntreprise.faces"*>

<h:outputText value=*"#{et.nom}"* />

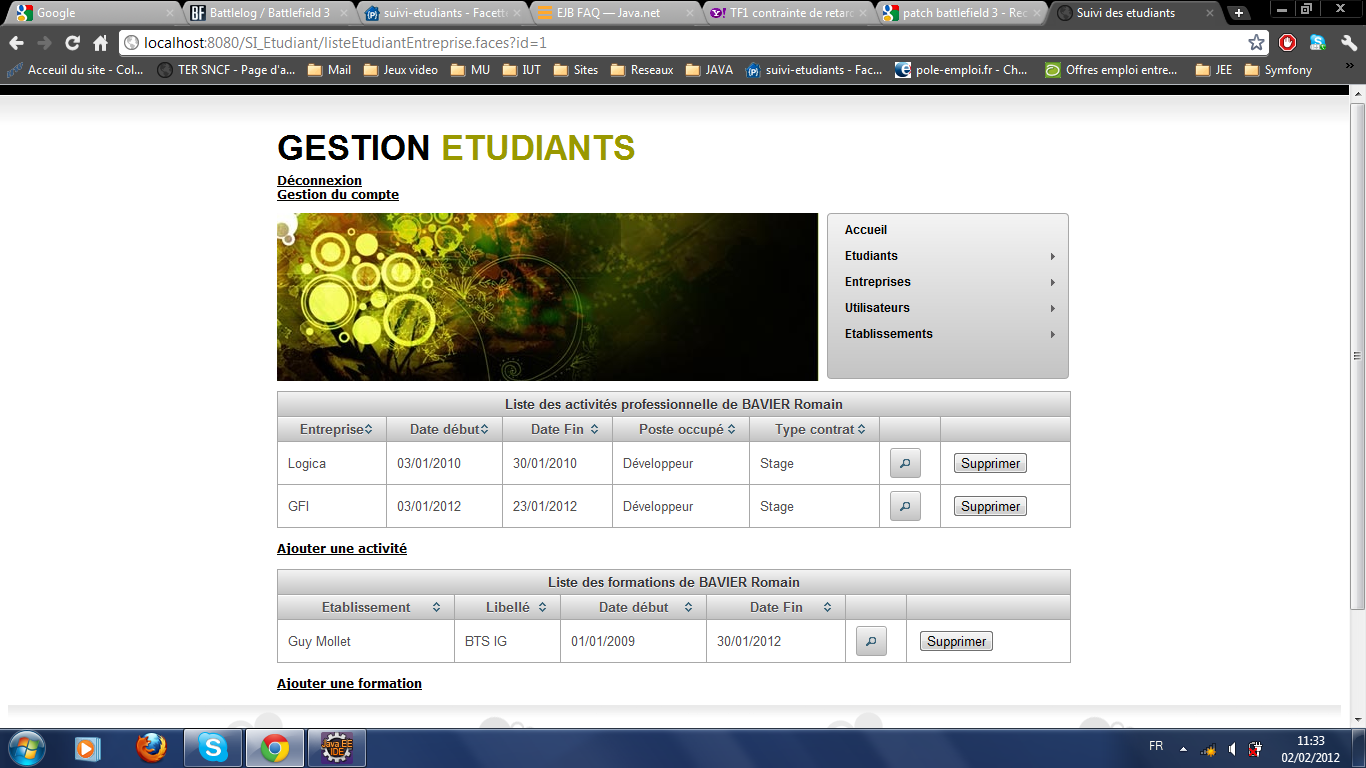
</h:outputLink>

</p:column>

value=*"#{etudiantBean.etudiants}"* va permettre de récupérer la liste etudiantsEntreprises.

Nous avons ensuite décidé d’y intégrer la librairie PrimeFaces. Cette librairie additionnelle offre de nouveaux composants plus riches à JSF, celui-ci s’avérant très limité. Les points forts de cette librairie sont surtout sa simplicité de mise en place, la richesse de ses composants et le fait qu’elle supporte l’AJAX. Elle permet d’ajouter des fonctions parfois très complexes d’apprentissage d’une manière beaucoup plus simple. Pour cela il suffit d’intégrer cette librairie (.jar) à notre projet.

Ce qui représente à l’écran.



\* Au niveau de l’Ajax voici ci-dessous le code permettant d’actualiser un combo box.

<h:selectOneMenu value=*"#{etudiantFormationBean.etablissementItemSelect}"*>

<f:selectItems value=*"#{etudiantFormationBean.etablissementsItems}"* />

<p:ajax listener=*"#{etudiantFormationBean.doSelectedEtab()}"* update=*"forma"* event=*"change"* />

</h:selectOneMenu>

<h:selectOneMenu id=*"forma"*value=*"#{etudiantFormationBean.formationItemSelect}"*>

<f:selectItems value=*"#{etudiantFormationBean.formationsItems}"* />

</h:selectOneMenu>

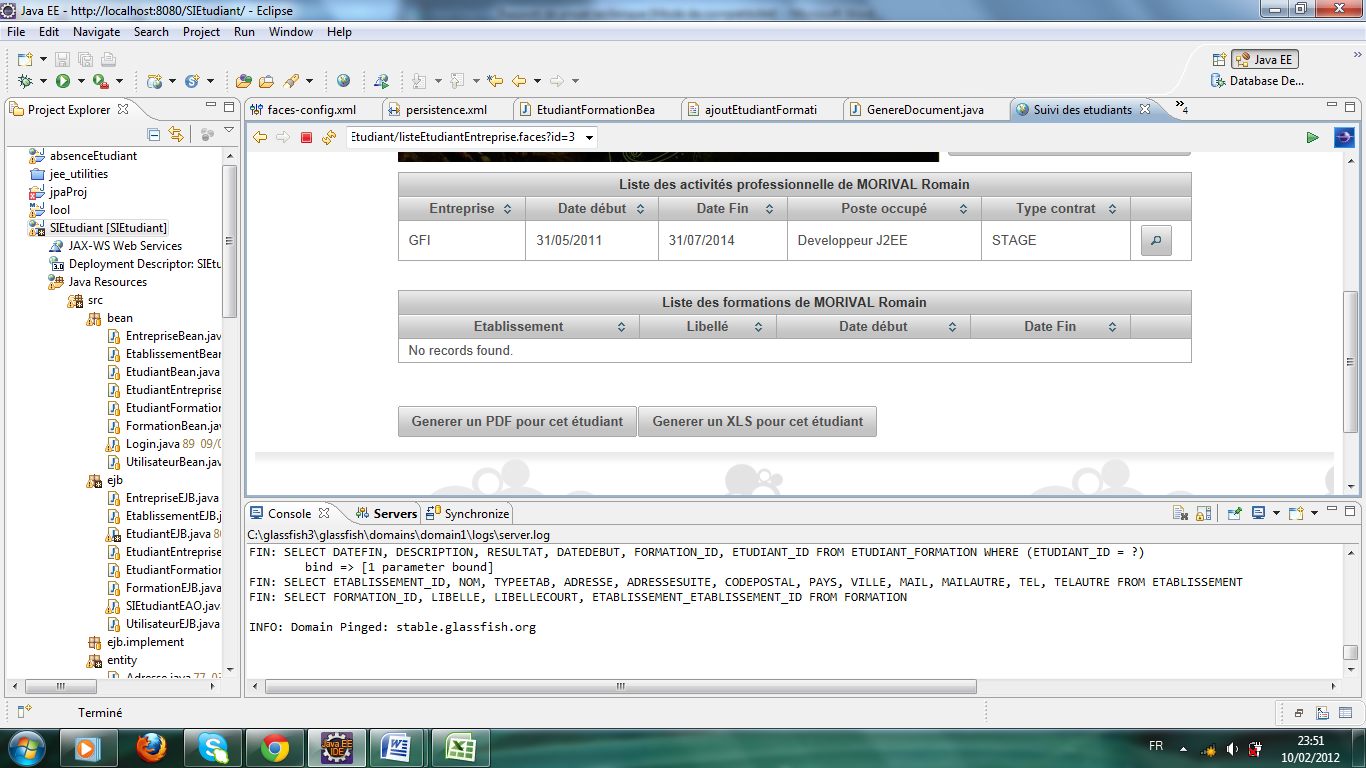
<p:ajax listener=*"#{etudiantFormationBean.doSelectedEtab()}"* update=*"forma"* event=*"change"* />

Lorsque l’on sélectionnera un établissement, cette instruction va enclencher automatiquement la fonction doSelectedEtab() du bean etudiantFormationBean, celle – ci renvoyant une liste de formation en fonction d’un établissement sélectionné.

update=*"forma"* event=*"change"* va permettre d’actualiser la liste des formationsItems de la combobox ayant pour id « forma ».

## 3.3 Ajout de fonctionnalité

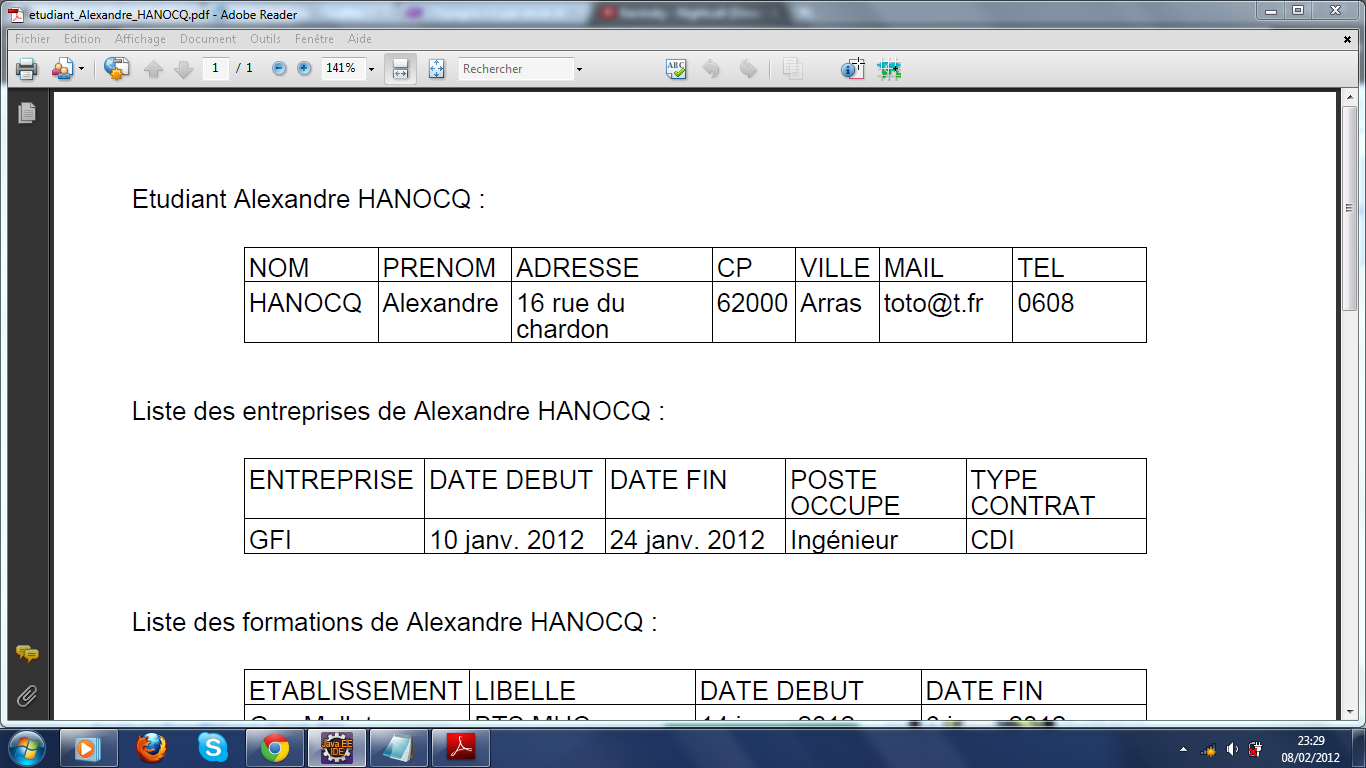
Nous avons ajouté certaines fonctionnalités dans le sujet du projet pour que celui-ci soit aussi complet que possible.



### 3.3.1 Génération de PDF avec la librairie itext.

L’impression des données nous paraissait primordiale pour l’utilisateur de l’application. Pour répondre à cela, nous avons opté pour la solution suivante qui consiste à générer un PDF, et comme tous les pdf, permet d’imprimer un document.

Nous avons utilisé iText, une API Open Source Java disponible sous AGPL.



Voici le code :

**public** **static** **void** creerListeEtudiantsPDF(List<Etudiant> etudiants,

String nomPDF) {

**try** {

// Flux de sortie -> Fichier

OutputStream file = **new** FileOutputStream(**new** File(

Util.*cheminFichier*(nomPDF, "pdf")));

// Creation d'un document

Document document = **new** Document();

// On recupere l'instance de la classe permettant d'ecrire et on

// lui passe le fichier et le document

PdfWriter.*getInstance*(document, file);

// Ouverture du document

document.open();

// Nom de l'auteur du PDF (celui qui utilise la machine)

document.addAuthor(Util.*userName*());

// Date de creation

document.addCreationDate();

// Titre du PDF

document.addTitle("Liste des etudiants");

// Tableau avec x-colonnes

PdfPTable table = **new** PdfPTable(7);

// Largeur des colonnes

table.setWidths(**new** **int**[] { 16, 16, 24, 10, 10, 16, 16 });

// Alignement des elements

table.setHorizontalAlignment(Element.*ALIGN\_CENTER*);

// Champs d'en-tete

table.addCell("NOM");

table.addCell("PRENOM");

table.addCell("ADRESSE");

table.addCell("CP");

table.addCell("VILLE");

table.addCell("MAIL");

table.addCell("TEL");

// Parcours de boucle, pour chaque etudiant de la liste

**for** (Etudiant etu : etudiants) {

// Ajouts

table.addCell(etu.getNom().toString());

table.addCell(etu.getPrenom().toString());

table.addCell(etu.getAdresse().getAdresse().toString());

table.addCell(etu.getAdresse().getCodePostal().toString());

table.addCell(etu.getAdresse().getVille().toString());

table.addCell(etu.getContact().getMail().toString());

table.addCell(etu.getContact().getTel().toString());

}

;

// Ajout du paragraphe au document

document.add(table);

// Fermeture du document

document.close();

// Fermeture du flux IO fichier

file.close();

// Ouvrir mon fichier

Util.*ouvrirFichier*(Util.*cheminFichier*(nomPDF, "pdf"));

// Attrapper les diverses exceptions, fichier, document et IO

} **catch** (FileNotFoundException fe) {

System.*out*.println(fe);

} **catch** (DocumentException de) {

System.*out*.println(de);

} **catch** (IOException ioe) {

System.*out*.println(ioe);

}

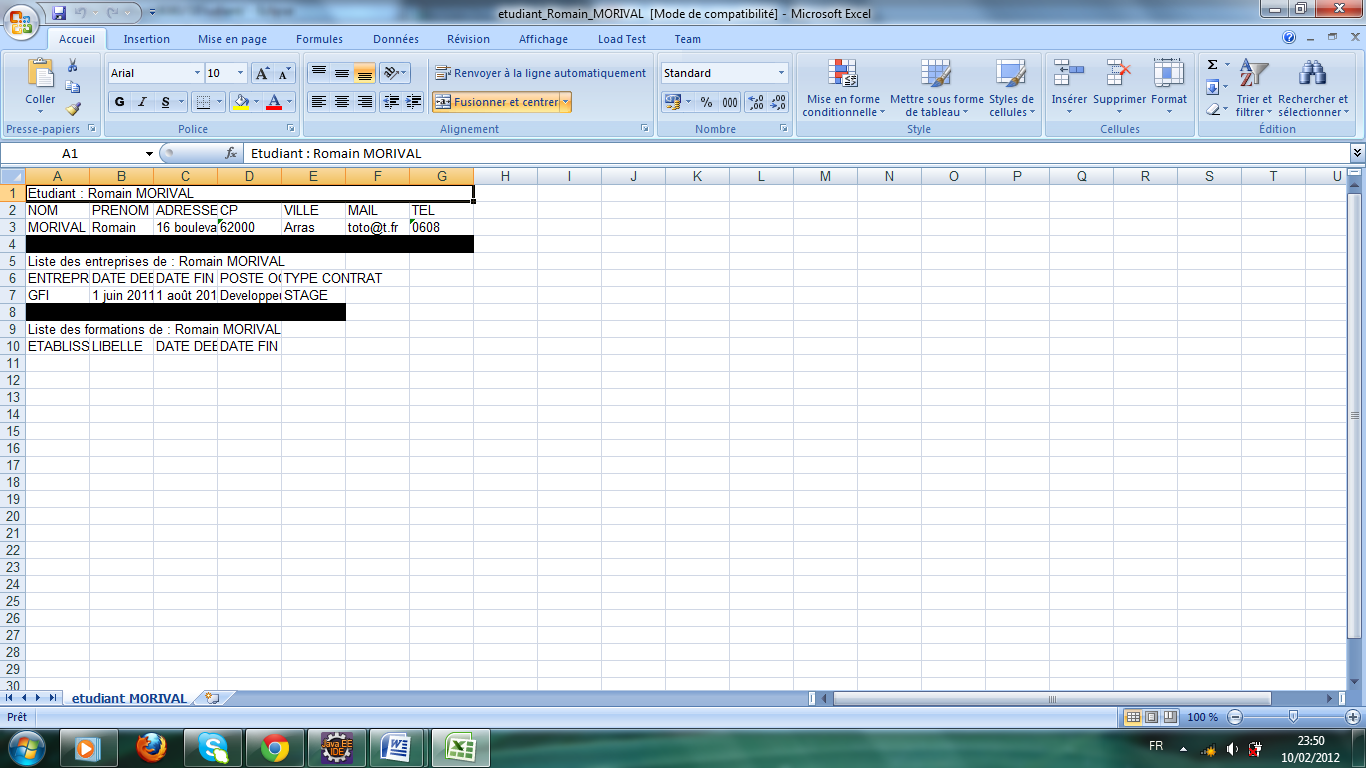
}

Explication du code

On passe la liste d’étudiants en paramètre…

### 3.3.2 Génération de document EXCEL avec la librairie POI.

En plus de PDF, nous avons opté pour la génération de document EXCEL en utilisant la librairie POI pour permettre à l’utilisateur de manipuler les données, comme effectuer des statistiques.



## 3.4 Sécurité

### 3.4.1 Cryptage des mots de passe.

**public** **static** String convertToHex(**byte**[] messageDigest) {

StringBuffer buf = **new** StringBuffer();

**for** (**int** i = 0; i < messageDigest.length; i++) {

**int** moitieByte = messageDigest[i] >> 4 & 0x0F;

**for** (**int** j = 0; j < 2; j++) {

**if**(moitieByte >= 10) {

buf.append((**char**) ('a' + moitieByte - 10));

}

**else** {

buf.append((**char**) ('0' + moitieByte));

}

moitieByte = messageDigest[i] & 0x0F;

}

}

**return** buf.toString();

}

/\*

\* Va crypter la chaine passée en parametre grace a la fct precedente

\*/

**public** **static** String crypter(String message) {

**try** {

*sha256Digest* = MessageDigest.*getInstance*("SHA-256");

} **catch** (NoSuchAlgorithmException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** *convertToHex*(*sha256Digest*.digest(message.getBytes()));

}

Afin de rendre l’application aussi sécurisée que possible, nous avons crypté les mots de passe à l’aide de la fonction crypter présente ci – dessus.

Ainsi avant l’ajout d’un utilisateur dans la base de données, le mot de passe sera préalablement passé dans cette fonction pour le convertir en chaine de caractére SHA256. Cette fonction n’a pas été écrite par nos soins.

### 3.4.2 Sécurité des pages.

Dans un souci de sécuriser au maximum les pages afin que celles – ci ne soient pas visibles par un utilisateur quelconque, nous avons fait en sorte d’appeler une fonction qui affiche ou non le contenu de la page. Voici le code présent dans chaque page JSF de notre application.

<ui:fragment rendered=*"#{login.isSuperAdmin()}"*>

//code de la page

</ui:fragment>

# Conclusion

Ce projet nous semble être une réussite car le travail réalisé correspond tout à fait aux attentes du sujet.

Le travail d’équipe n’a posé aucun problème, une bonne entente de la part de chacun, des idées partagées nous ont permis de contribuer à l’aboutissement du projet.