**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](#_Toc316681791)

[2. Cahier des charges 3](#_Toc316681792)

[2.1. Objectifs 3](#_Toc316681793)

[2.2. Contraintes technologiques 3](#_Toc316681794)

[2.3 Analyse des Besoins et MCD 3](#_Toc316681795)

[3. Partie Technique 4](#_Toc316681796)

[3.1 Structure générale de l’application 4](#_Toc316681797)

[3.2 Le MVC 4](#_Toc316681798)

[3.2.1. Le Modèle 4](#_Toc316681799)

[3.2.1.1. Les Entités 4](#_Toc316681800)

[3.3.1.2. La couche d’accès aux données 4](#_Toc316681801)

[3.2.2 La couche contrôle 4](#_Toc316681802)

[3.2.3 La couche vue 4](#_Toc316681803)

[3.3 Ajout de fonctionnalité 4](#_Toc316681804)

[3.3.1 Génération de PDF avec la librairie itext. 4](#_Toc316681805)

[3.3.2 Génération de document EXCEL avec la librairie POI. 4](#_Toc316681806)

[3.4 Sécurité 4](#_Toc316681807)

[3.4.1 Cryptage des mots de passe. 4](#_Toc316681808)

[3.4.2 Sécurité des pages. 4](#_Toc316681809)

[3.4.3 Test unitaire. 4](#_Toc316681810)

[4. Conclusion 4](#_Toc316681811)

# Introduction

Afin 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 devions, Romain BAVIER, Alexandre Hanock et Gerardo Mootinal réaliser un projet informatique d’une durée de six semaines. 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 de l’IUT. Les utilisateurs de l’application seront des personnes internes à l’IUT tels que les enseignants, personnel d’administration, et auront suivant leur droit, de visualiser, modifier, ajouter des informations sur les étudiants ainsi que sur leur suivi.

## 2.2. Contraintes technologiques

Les technologies utilisées pour ce projet furent imposée dans le sujet du projet. Ces technologies étaient J2EE6 associé au Framework (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) JSF 2.0.

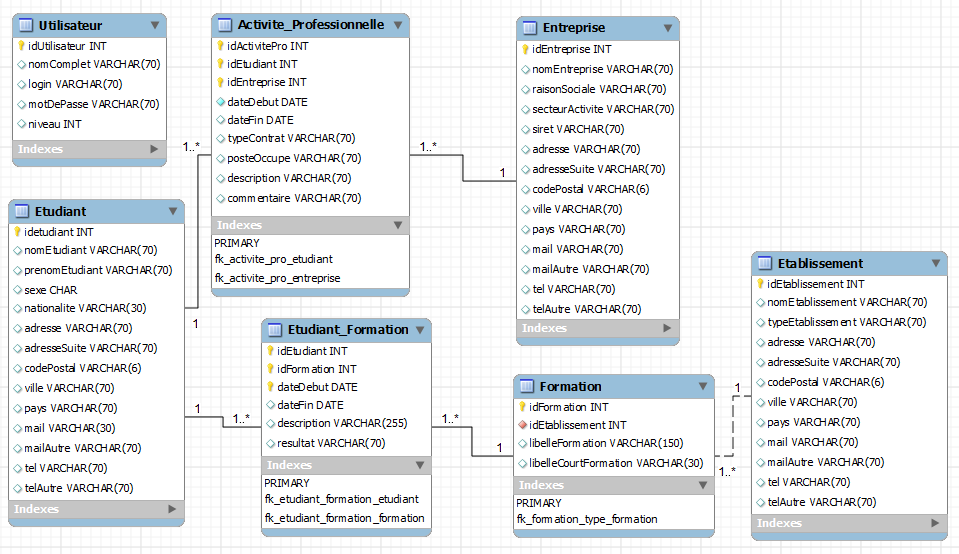
Pour ce qui est de la persistance de nos données, nous avons choisit eclipseLink, Framework open source de mapping objet-relationnel, fournissant une plateforme puissante et flexible permettant de stocker des objets Java dans une base de données relationnelle. Il est un dérivé de TopLink de la société 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 somme passé sur le SGSBD POSTGREY.

Enfin le serveur d’application utilisé était glassfish.

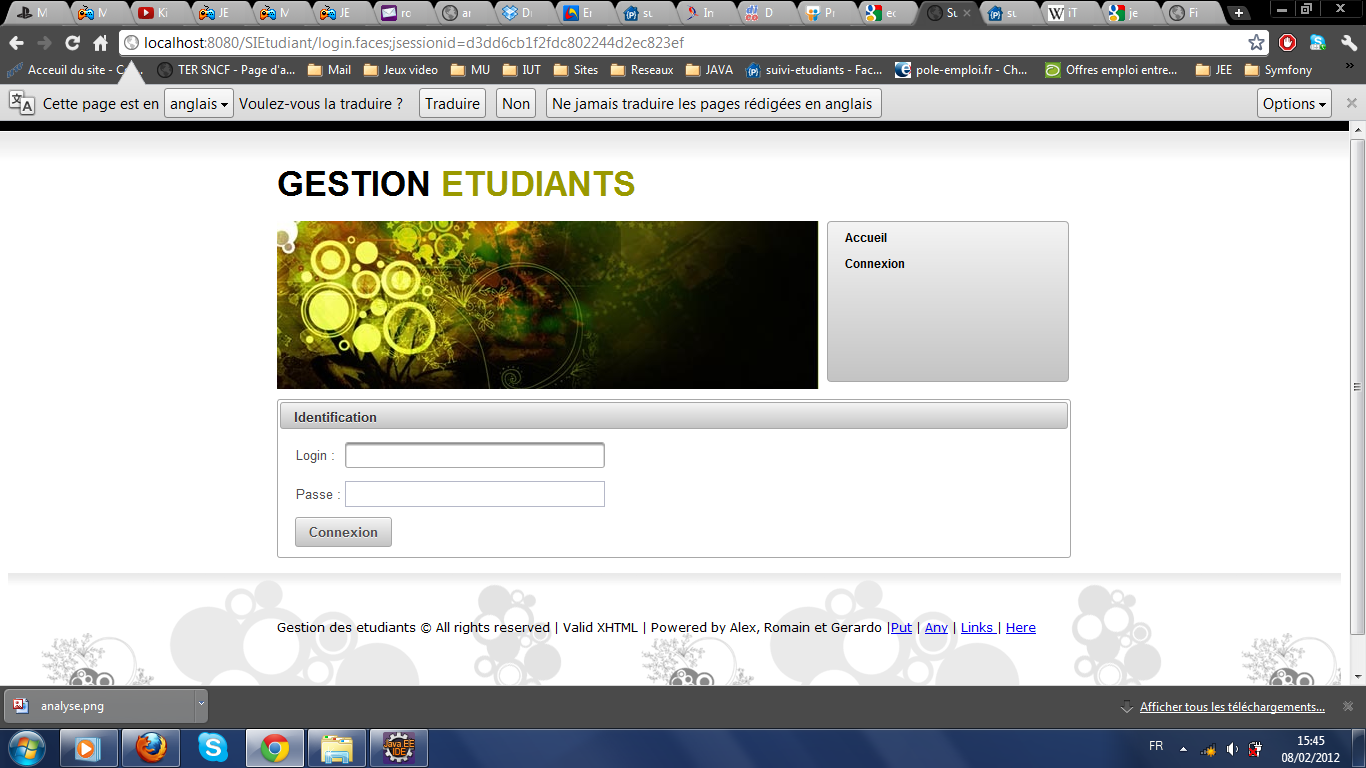
## 2.3 Analyse des Besoins et MCD

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



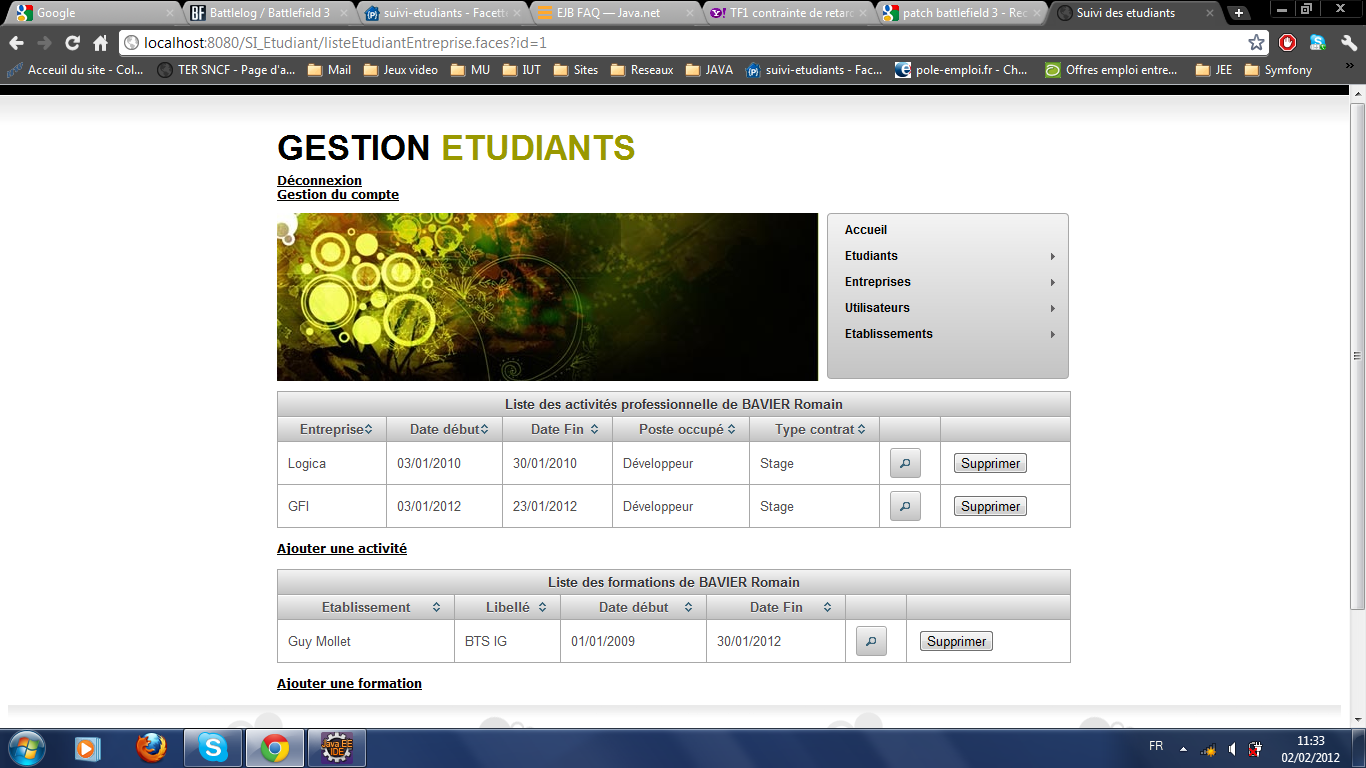
# Partie Technique

## 3.1 Structure générale de l’application



*Page de login (Non connecté)*

Ci-dessus, cette page est la page principale de l’application. Elle 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 accède à l’espace de gestion. Il pourra accéder, par exemple ci-dessus à liste des activités de l’étudiant qui aura sélectionné au préalable avant.

En fonction du rôle de l’utilisateur, le menu à droite affiche ou non les sous menu. Ce menu permet la navigabilité 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 s’épare les données (le modèle), l'interface homme-machine (la vue) et la logique de contrôle (le contrôleur).

### 3.2.1. Le Modèle

Cette couche représente  les données de l'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 doit obligatoirement :

- avoir un constructeur sans argument et être marquée de l'annotation @Entity.

- posséder au moins une propriété déclarer comme clé primaire avec l'annotation @Id,

- implémenter *Serializable* processus de sauvegarde d'un objet complet, d'où il pourra être restauré à tout moment.

Une entité est composée de propriétés qui seront mappés 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).

Donc chaque table de notre base de données possède son entité. Les entités de notre code sont plus ou moins identique sauf quand la clé primaire est sur plusieurs champs.

Ci-dessous, voici 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;

}

@Column(length = 70)

**public** String getNom() {

**return** nom;

}

**public** **void** setNom(String nom) {

**this**.nom = nom;

}

@Column(length = 70)

**public** String getPrenom() {

**return** prenom;

}

**public** **void** setPrenom(String prenom) {

**this**.prenom = prenom;

}

@Embedded

**public** Adresse getAdresse() {

**return** adresse;

}

@Embedded

**public** Contact getContact() {

**return** contact;

}

**public** **void** setContact(Contact contact) {

**this**.contact = contact;

}

**public** **void** setAdresse(Adresse adresse) {

**this**.adresse = adresse;

}

**public** String toString() {

**return** nom + " " + prenom;

}

}

A noté que les annotations présentent 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.

@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 etudiant\_formation.java.

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

#### 

#### 3.3.1.2. La couche d’accès aux données

Cette couche permet l’accès aux données de la base de données.

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

@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);

}

/\*

\* Cherche et retourne un etudiant par son identifiant

\*/

**public** Etudiant findEtudiantById(Long id) {

Etudiant results = (Etudiant) em.find(Etudiant.**class**, id);

**return** results;

}

@SuppressWarnings("unchecked")

**public** List<Etudiant> findAllEtudiantsByEnt(Long ent) **throws** EJBException {

List<Etudiant> results = em

.createQuery(

"select DISTINCT e from Etudiant e, EtudiantEntreprise ent" +

" where e.id=ent.id.etudiantId and ent.id.entrepriseId=:ent")

.setParameter("ent", ent).getResultList();

**return** results;

}

}

Nous avons utilisé les 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 JEE.

Cette architecture propose un cadre pour créer des composants distribués hébergé 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 sql mais JPQL. Dans ce type de requête, on ne manipule pas la table directement mais les classe entités.

En résumé de notre ejb,

[PersistenceContext](http://java.sun.com/javaee/5/docs/api/javax/persistence/PersistenceContext.html)(unitName = « si\_etu »), permet à l'EJB de récupèrer une instance de l'interface [EntityManager](http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html). Celle-ci va permettre les opérations de persistance sur les entités, tels que create, select, update et delete. 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 contenu dans le répertoire META-INF, répertoire contenant les métas donnés, c'est-à-dire les fichiers descripteurs de déploiement.

<?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>

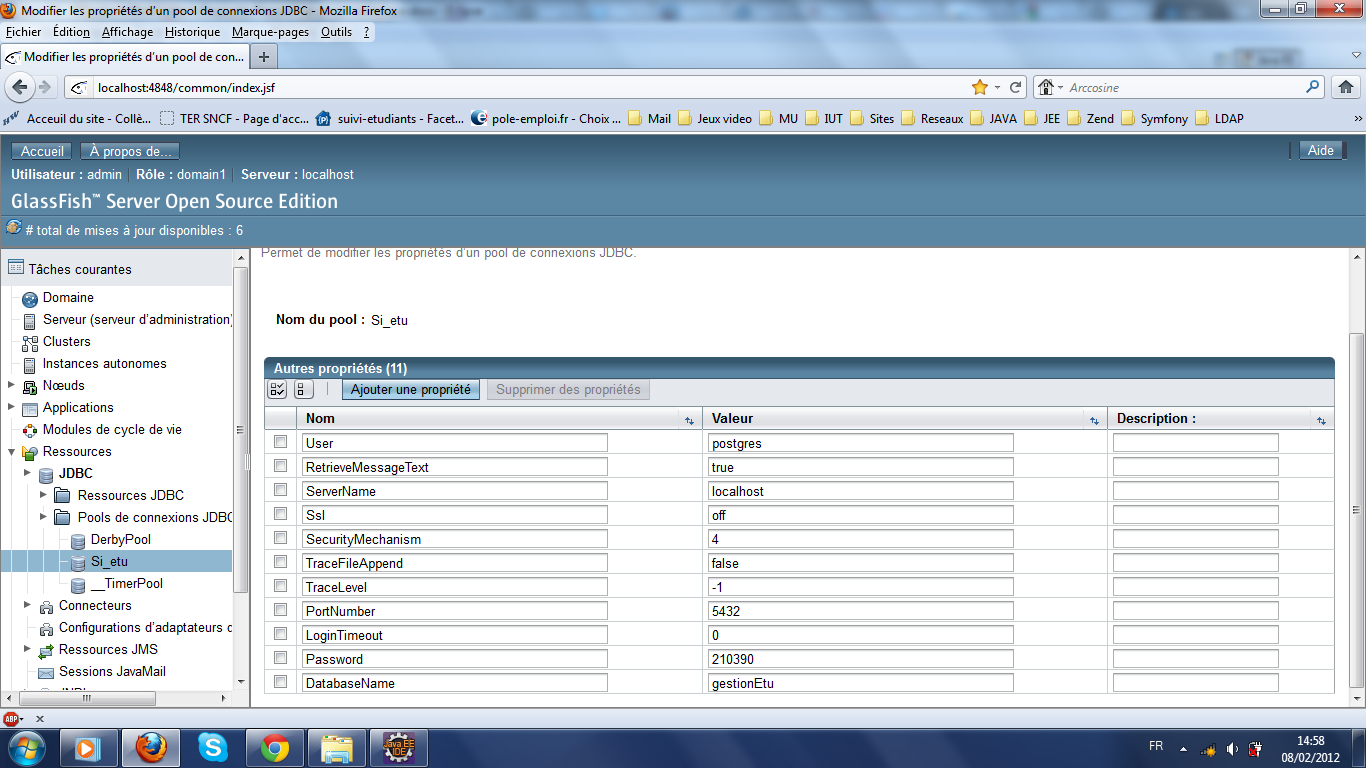
<property name=*"eclipselink.logging.level"* value=*"FINE"* />

</properties>

</persistence-unit>

</persistence>

La ligne Jdbc/gestionEtu 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 configurer tout cela dans la console d’administration du serveur d’application glassfish.



### 3.2.2 La couche contrôle

Cette couche fait une sorte de passerelle entre l’utilisateur et les données.

Nous n’avons pas configuré cette couche dans un fichier managedBean.xml, mais choisie de mettre des annotations directement dans les fichiers JAVA concernés.

Pour que la vue accède aux attributs du contrôleur, la présence de getters est nécessaire.

C’est dans cette couche que l’on définit les fonction 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 => retourne list (voir

// face-config.xml)

**public** String modifier() {

etudiantEJB.updateEtudiant(editEtudiant);

conversation.end();

**return** "listEtudiant";

}

// fonction retournant edit (voir face-config.xml)

**public** String edit() {

**return** "edit";

}

}

// getter and setter

En ce qui concerne les scopes du framework JSF utilisés, nous avons utilisé pour pratiquement tout 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 compatible avec ajax.

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

L’annotation Postconstruct est utilisée sur une méthode qui doit être éxécuté après l’injection de dépendance effectuée lors d’une initialisation. Cette méthode doit être invoquée devant la classe.

### 3.2.3 La couche vue

Cette couche correspond à l’affichage de notre site. Nous avons donc utilisé dans un premier temps les balises facelets du framework JSF 2.0 (langage EL).

*Facelets* est une technologie de présentation pour le développement d'applications web en *Java*. Une page *JSP* est 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.

*Facelets* est basé 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 la couche contrôle 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.

Une fois le projet bien avancer, nous avons intégrer la librairie PrimeFaces à notre projet. 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 à mettre en place, la richesse de ses composants et le fait qu’elle supporte l’AJAX.

\* 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"* />

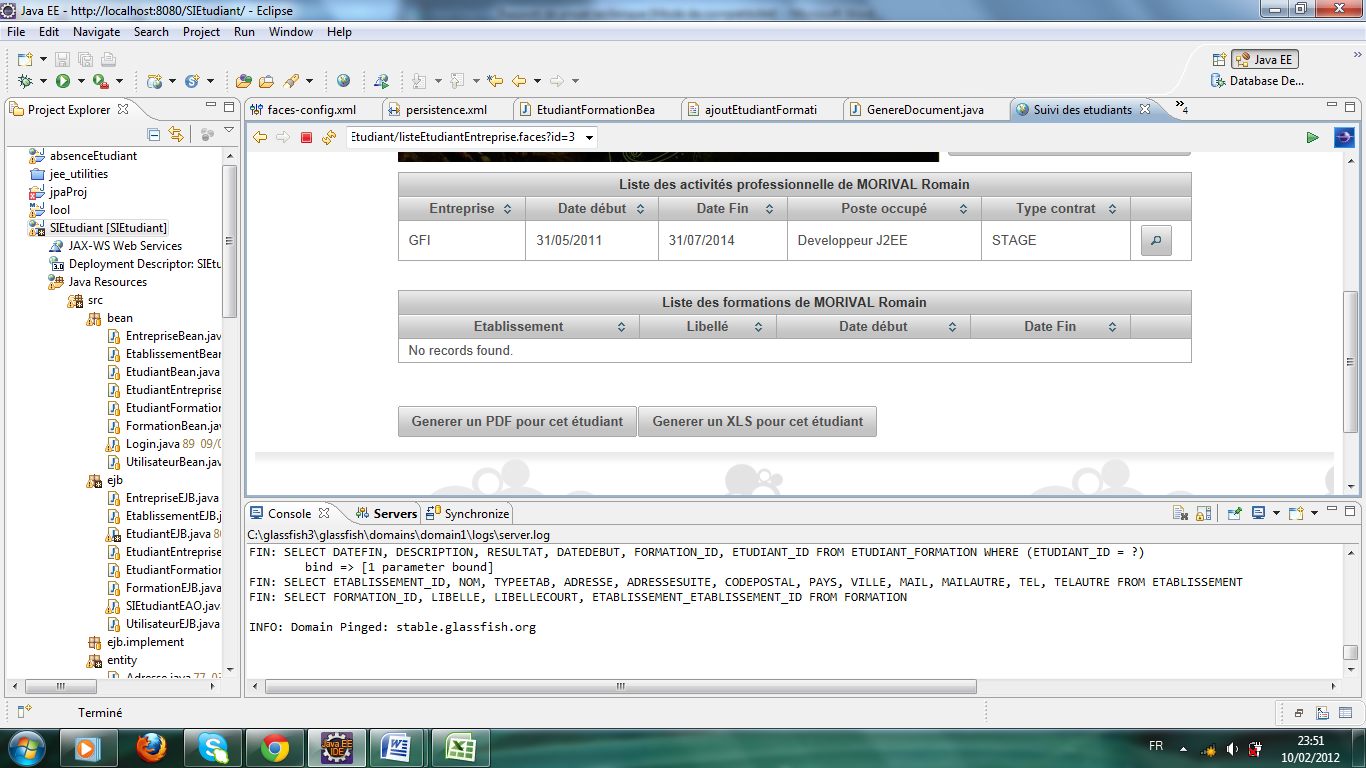
Lorsque l’on sélectionnera un établissement, cette instruction va enclencher automatiquement la fonction doSelectedEtab() du bean etudiantFormationBea, celle – ci renvoyant une liste de formation en fonction d’un établissement sélectionné.

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

\* Au niveau du design

## 3.3 Ajout de fonctionnalité

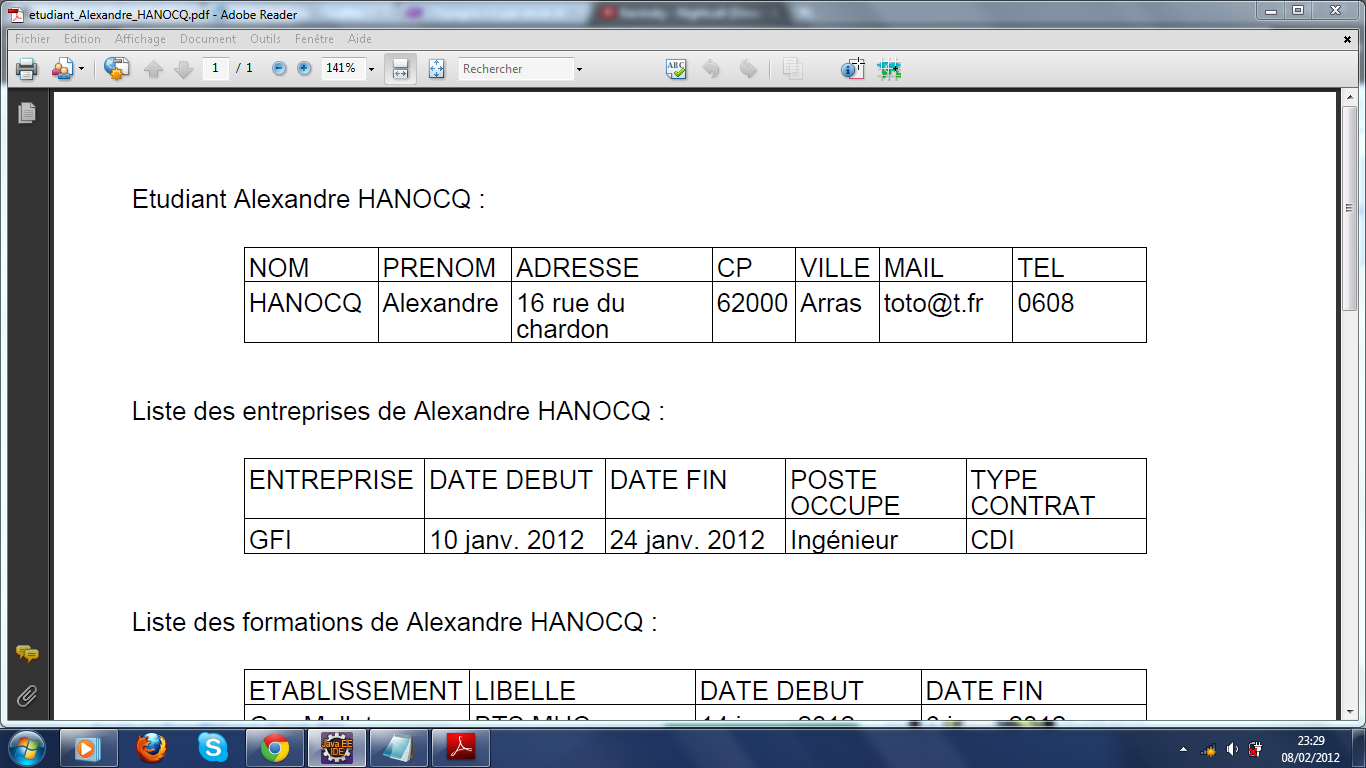
Nous avons ajouté certaines fonctionnalités dans le sujet du projet pour que celui-ci soit aussi complet que possible.



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

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

Nous avons utiliser iText, une API Open Source Java disponible sous AGPL, car alexandre l’avais déjà utiliser.



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);

}

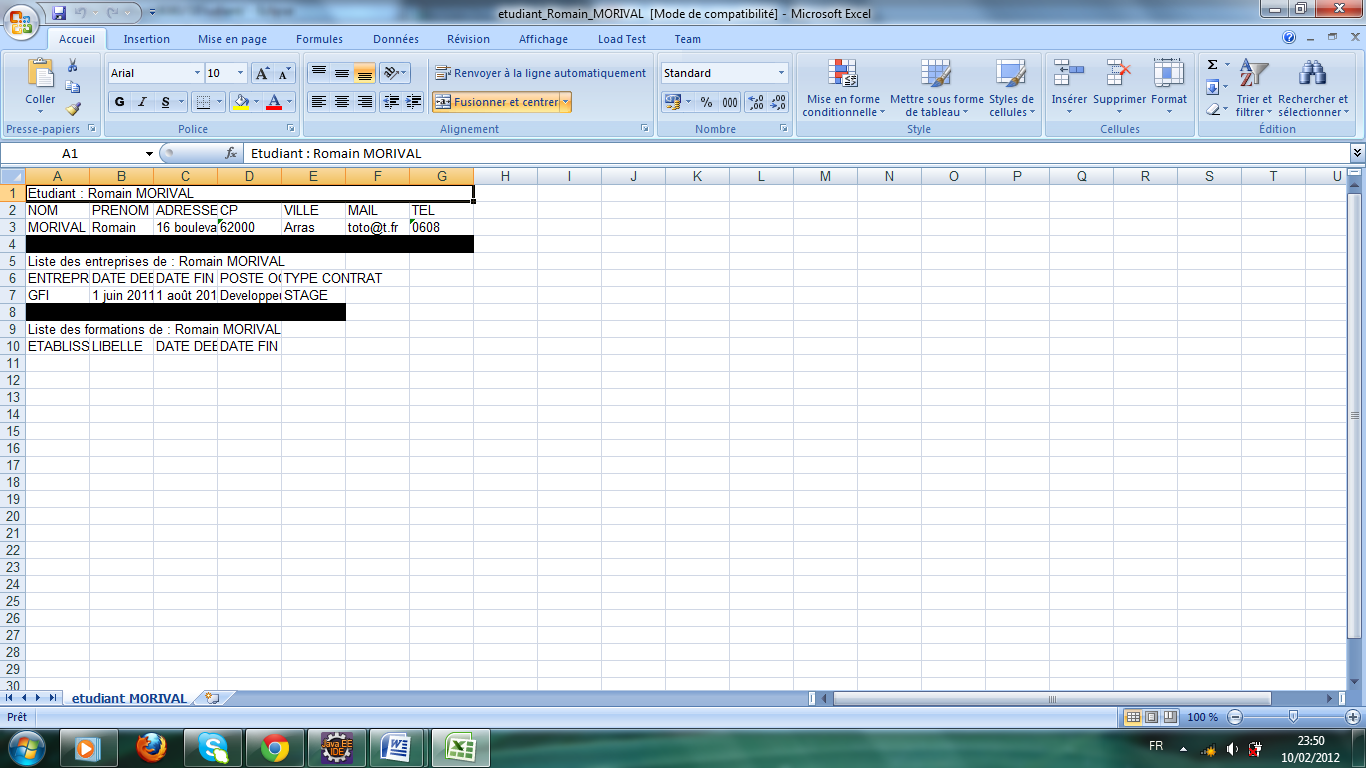
}

Explication du code

On passe la liste d’étudiant 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.



## 3.4 Sécurité

### 3.4.1 Cryptage des mots de passe.

Fonction de cryptage

### 3.4.2 Sécurité des pages.

### 3.4.3 Test unitaire.

Code test unitaire (quand ils marcheront)

# Conclusion

Le projet est une réussite. Le travail réalisé correspond tout à fait aux attentes du sujet.

Le travail d’équipe n’a posé aucun problème, chacun de nous à contribuer au projet. Il nous a apporté de nouvelles connaissances.