Projet : J2EE
Rapport final

Fabian Guion Lorraine Perronnet Audric Schiltknecht
Denis Villand

Année 2009

Résumé

Implémentation d'une petite application de commerce électronique en utilisant les technologies *Java* JEE.

Table des matières

1 Objectif

Le but de ce projet est de réaliser un site de commerce électronique mettant en application les notions vues en cours et TPs de JEE (EJB 3, Servlets, JSP...) ainsi que des technologies aditionnelles non étudiées.

Le schéma suivant était suggéré :

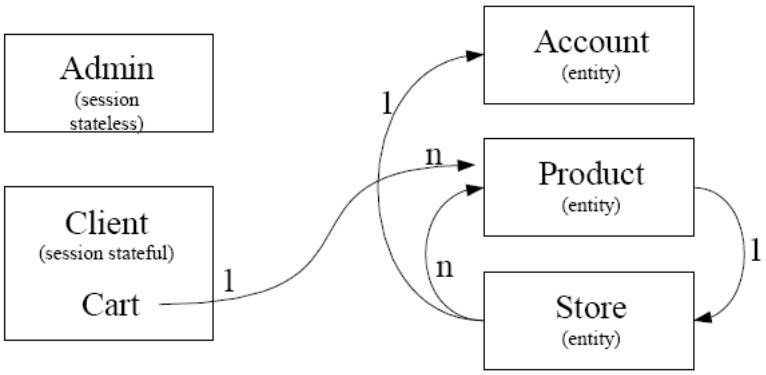

Fig. 1 – Architecture suggérée

2 Notre Site : eBide

Pour répondre à la problématique posée, nous nous sommes portés sur un site d'enchère : eBide. Le principe était d'en faire un site permettant aux grands de ce monde d'effectuer des transactions peu orthodoxes, un lieu rêvé pour mettre en vente un lot de Rafales à prix cassés à destination des pays Moyen-Orientaux, par exemple.

2.1 Actions

Un utilisateur devait donc pouvoir créer un compte, se connecter, mettre en vente un produit de son choix, renchérir sur un produit qui l'intéresse, gérer ses enchères en cours, etc... Pour le détail des fonctionnalités du site, on se reportera au schéma 7 de la section « Vue » (section 5.2.2), qui présentent l'enchainement des différentes pages.

2.2 Cycle de vie d'une vente

Un utilisateur décide de mettre en vente un objet sur notre site. Le diagramme 2 décrit les différentes étapes de la vie de cet objet, depuis la mise en vente, jusqu'à la réception (ou la non-vente) de l'objet.

3 Déroulement du projet

3.1 Répartition

La répartition théorique des tâches était la suivante :

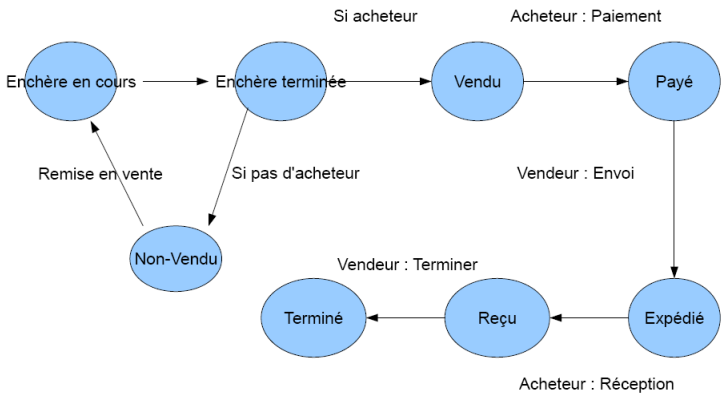

FIG. 2 – Cycle de vie

Fabian G. : Pages jsp

Lorraine P. : Réalisation de la CSS, pages jsp

Audric S. : Réalisation du back-end

Denis V. : Connexion back-end/front-end

En pratique, nombre de tâche se sont entrecroisées, et chacun d'entre nous a pu avoir une bonne vision du projet dans son ensemble.

3.2 Organisation

Nous avons en début de projet, établi le déroulement décrit sur la figure 3.

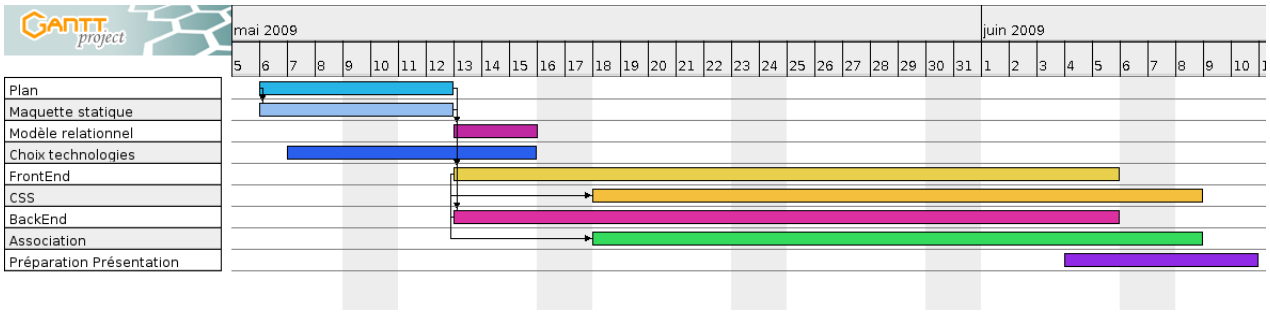

FIG. 3 – Déroulement du projet

3.3 Problèmes rencontrés

Nous avons eu quelques petits problèmes avec JBOSS, qui semblait certaines fois ne pas apprécier l'EAR crée. Cependant, un redémarrage du serveur, et un nettoyage du dossier *deploy* suffirent la plupart du temps à surmonter cet obstacle.

Lorsque nous avons ajouté des librairies aditionnelles, il nous est arrivé que JBOSS se plaigne de ne pouvoir lire les *TLD* (fichiers de descriptions des nouveaux *tag*). Comme précédement, un redémarrage du serveur suffisait à régler ce problème.

4 Technologies utilisées

4.1 JSF

4.1.1 Présentation

Le site developpez.com donne la définition suivante :

> Java ServerFaces (JSF) est un framework de développement d'applications Web basé sur les composants, qui permet de développer des interfaces client riches tout en respectant le paradigme MVC.

Concrètement, JSF est une spécification (actuellement en version 1.2) pour fournir un ensemble d'API et de jeux de composants pour le développement d'applications web similairement au développement d'application « lourdes » en Swing par exemple.

Implémentation choisie : Pour des raisons de simplicités, nous avons décidé d'utiliser l'implémentation fournie[1] dans JBoss, nommée Mojorra.

4.1.2 Librairies additionnelles

Les composants fournis par JSF s'avèrent néanmoins limités. En effet, s'il est tout à fait possible de développer des applications web « classiques » c'est-à-dire ayant l'apparence habituelle des pages que l'on peut trouver sur l'Internet (formulaire, cases à cocher, liens, etc.), il n'y a aucun composant apportant des effets plus poussé. C'est là qu'interviennent les librairies additionnelles.

De telles librairies apportent des composants proposant une apparence et un comportement améliorés, tels que les propositions de réponses dans un champ de recherche, des tableaux avec fonctionnalité de tri automatique, et autres joyeusetés.

Choix de la librairie additionnelle : Il existe de nombreuses librairies proposant des composants additionnels : RichFaces, ICEFaces, Tomahawk, etc.

Notre choix s'est initialement porté sur RichFaces, jeux de composants développé par l'équipe de JBoss. Malheureusement, suite à des problèmes d'incompatibilités[2], nous avons dû changer notre décision. C'est alors que nous avons découvert OpenFaces, une toute récente librairie, en plein développement, et ayant l'avantage de ne nécessiter que très peu de dépendances, et donc légère.

4.1.3 Avantage

Le choix de la technologie JSF nous a permis de concevoir notre site en suivant le modèle MVC, puisque la séparation de la Vue et du Controlleur peut être facilement faite. Le Modèle de notre application est la partie BackEnd, avec les EJB et la gestion de la base de donnée. Les classes du Controlleur sont dans le package controller, et la partie Vue se résume aux pages jsp, dans WebContent.

[1] C'est aussi l'implémentation fournie et maintenue par Sun

[2] Problème avec la suggestion de recherche

4.2 De l'internationalisation du site

Pour se conformer au côté « politique internationale » que nous voulions donner au site, nous avons fait en sorte que celui ci soit disponible en 2 langues (français/anglais), et que l'utilisateur puisse basculer de l'une à l'autre en cours de navigation. La gestion des *bundles* offerte par `JSF` nous a facilité la tâche. Le principe est qu'au lieu d'avoir effectivement 2 versions de chaque page (une par langue), on stocke tous les textes à afficher dans des fichiers à part (les message bundles) et c'est au moment de la génération de la page que le choix du texte sera fait, en fonction de la langue en cours.

Un exemple simple : le titre de la page courante (par exemple index.jsp). On veut afficher « eBide : Accueil » en français, « eBide : Welcome » en anglais. On va créer 2 fichiers, les bundles, pour chaque langue : index_en.properties et index_fr.properties (package org.eBide.message du FrontEnd). Dans chacun d'eux, on associera un tag à un certain nombre d'éléments textuels (ceux que l'on voudra afficher dans la page .jsp). Il restera à déclarer ce bundle dans faces-config.xml pour pouvoir s'en servir.

Reprenons notre exemple :

On écrit les fichiers .properties :

```
current_page=index
page_title=eBide - Accueil
presentation=Bienvenue sur eBide.
...
```

Listing 1 – index_fr.properties

```
current_page=index
page_title=eBide - Welcome
presentation=Welcome on eBide
...
```

Listing 2 – index_en.properties

On déclare le bundle dans faces-config.xml :

```
<application>
  ...
  <message-bundle>org.eBide.message.index</message-bundle>
  ...
</application>
```

On associe un tag spécifique au bundle, puis on l'utilise dans index.jsp :

```
<o:loadBundle basename="org.eBide.message.index" var="page"/>
...
<title><h:outputText value="#{page.page_title}"/></title>
```

Si le site est en anglais, la variable **page** fera référence au index_en.properties, et de même en français. On a ainsi toujours la langue désirée.

5 Conception du site

5.1 BackEnd

Le backend contient le code métier de l'application : c'est ici où l'on définira les entités manipulées et qui seront persistées en base de données, ainsi que les diverses interfaces permettant de stocker, récupérer et modifier ces entités.

5.1.1 Le modèle

Le modèle de notre application a été réalisé en suivant les spécifications des `EJB 3`. La figure 4 présente le modèle relationnel initial : en bleu sont représentées les classes que nous avons développées et utilisées, et en jaune, celles que nous avons identifiées dans la modélisation mais que nous n'avons pu coder.

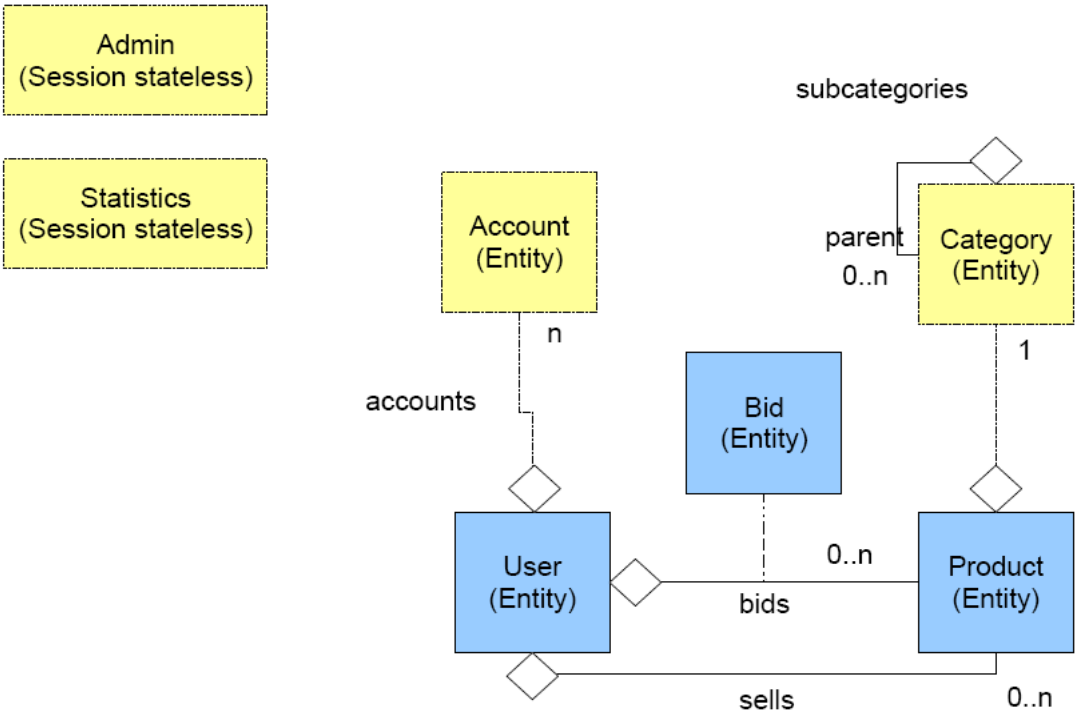

FIG. 4 – Modèle relationnel

5.1.2 Les interfaces

Nous avons définis certaines interfaces qui permettent d'accéder depuis la vue au entités stockées en base de donnée. Ces interfaces effectuent les requêtes nécessaires, en s'appuyant sur le langage `EJB QL`.

Ces requêtes sont mises à disposition de la vue *via* des interfaces annotées @REMOTE.

5.2 FrontEnd

Le frontend se divise en deux grandes parties : le controleur et la vue.

5.2.1 Controleur

Les beans managés : Par soucis de lisibilité, le controleur a été fragmenté en plusieurs beans managés. Les ProductController, UserController, et BidController regroupent chacun les méthodes permettant la gestion des classes associées côté modèle. Ils se subtituent ainsi aux servlets traditionnels, les redirections entre les pages étant gérées par le fichier faces-config.xml. Le ProductController permet par exemple de créer un produit, de l'éditer, de récupérer la liste des produits suivis par un utilisateur, etc... à partir des pages jsp.

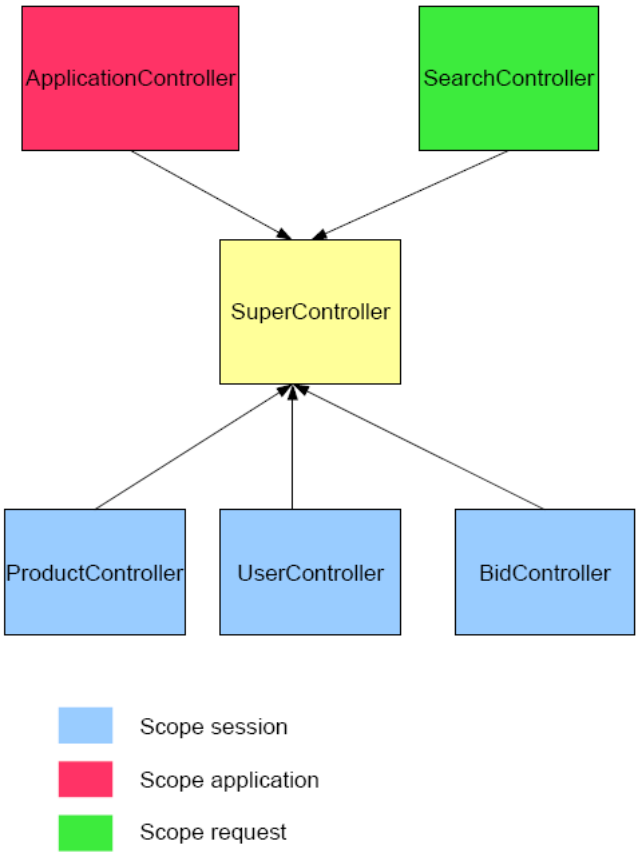

Fig. 5 – Les classes du controleur

Tous les controleurs étendent la classe abstraite SuperController. Cette dernière permet en particulier d'accéder aux fonctions du BackEnd via des attributs annotés « `@EJB` ». Par exemple, **bidManager** fait ainsi automatiquement référence à une instance de ManageBidRemote, côté Modèle. De plus, il est souvent nécessaire, au sein d'un contrôleur, d'avoir accès aux informations d'un autre contrôleur. Par exemple, la méthode `getBiddenProducts`, quoiqu'attribuée au ProductController, a besoin d'informations sur l'utilisateur courant et donc d'avoir accès à l'instance de UserController dans la session courante. Le SuperControlleur dispose donc des accesseurs permettant aux différents conrtôleurs « spécialisés » d'interagir.

Lorsqu'un utilisateur est connecté, il est nécessaire de conserver un certain nombre d'informations spécifiques pendant toute la durée de la session, que ce soit ses paramètres

utilisateurs, ses produits mis en vente ou ses enchères. Trois controleurs appartiennent ainsi au scope session : UserController, ProductController et BidController. Chacun d'eux est à priori destiné à interagir avec l'entité correspondante côté BackEnd. Toutes les tâches impliquant un User seront ainsi traitées par le UserController (création d'un compte, connexion, édition d'un profil...) La gestion de la langue de navigation revient également à l' UserController (c'est une donnée que l'on souhaite persistante pour la durée d'une session, et pas simplement pour une page...).

Dans le scope application, et comme son nom l'indique, on retrouve le ApplicationController. Il traite les requêtes dont le résultat ne dépend pas de l'état de l'utilisateur courant (connecté ou pas). On n'a donc pas besoin de se placer en scope session. On y retrouvera, par exemple, une méthode pour obtenir la liste des n dernières mises en vente, ou pour retourner la liste des noms de catégories de produits dans la langue adaptée...

Enfin, dans le scope request, seul le SearchController est défini. Sa principale fonction est de réaliser une recherche dans la base de donnée en BackEnd, puis d'en retourner le résultat. La durée de vie des informations mises en jeu ne dépasse pas celle de la requête (le résultat de la recherche est envoyé à la vue mais n'est pas conservé par la suite...). Un passage en scope session -ou en scope application- ne s'impose donc pas ici.

Les composants : En JSF, on retrouve d'autres composants au niveau controller, bien qu'ils ne fassent pas partie des beans managés que nous avons définis comme « controleur » précédemment. Les Validators, par exemple, permettent de vérifier certaines propriétés sur le contenu d'un formulaire avant de le soumettre (égalité du mot de passe de confirmation avec le mot de passe d'origine dans le cadre d'une inscription, par exemple). L'ensemble de nos Validators se trouve dans le package org.eBide.validator. On citera aussi les Converters, permettant de convertir une donnée émanant d'un formulaire avant de la soumettre au controleur. C'est via un Converter, par exemple, que nous avons réalisé l'encryption du mot de passe dans notre projet (le mot de passe est « crypté[3] » avant toute transmission par le PasswordEncryptor, qui est un simple Converter placé dans org.eBide.converter).

faces-config.xlm : Ce fichier occupe un rôle central dans la partie controleur d'une application réalisée en JSF. C'est ici que sont répertoriés les beans managés et tous les autres composants (validators, converters, mais aussi tous les bundles, que nous avons utilisés en particulier pour le passage français - anglais du site). On y retrouvera également la gestion des règles de navigation du site, en l'absence de servlets redirigeant les requêtes vers d'autres pages. Par exemple :

```
<navigation-rule>
  <from-view-id>/registration.jsp</from-view-id>
  <navigation-case>
    <from-outcome>creationFailed</from-outcome>
    <to-view-id>/failure.jsp</to-view-id>
  </navigation-case>
```

[3]Nous n'avons utilisé aucune fonction de hashage, juste un basique ajout d'une chaîne de caractère en tête du mot de passe pour démontrer la possibilité d'utiliser des mots de passe cryptés

```
    <navigation-case>
      <from-outcome>connectionOK</from-outcome>
      <to-view-id>/index.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
```

On définit les règles de navigation depuis la page registration.jsp. Si la création du compte se passe bien, la méthode dédiée du UserController renverra connectionOK, et l'utilisateur sera alors redirigé vers l'index. Dans le cas contraire, faces-config.xml le renverra vers une page d'erreur...

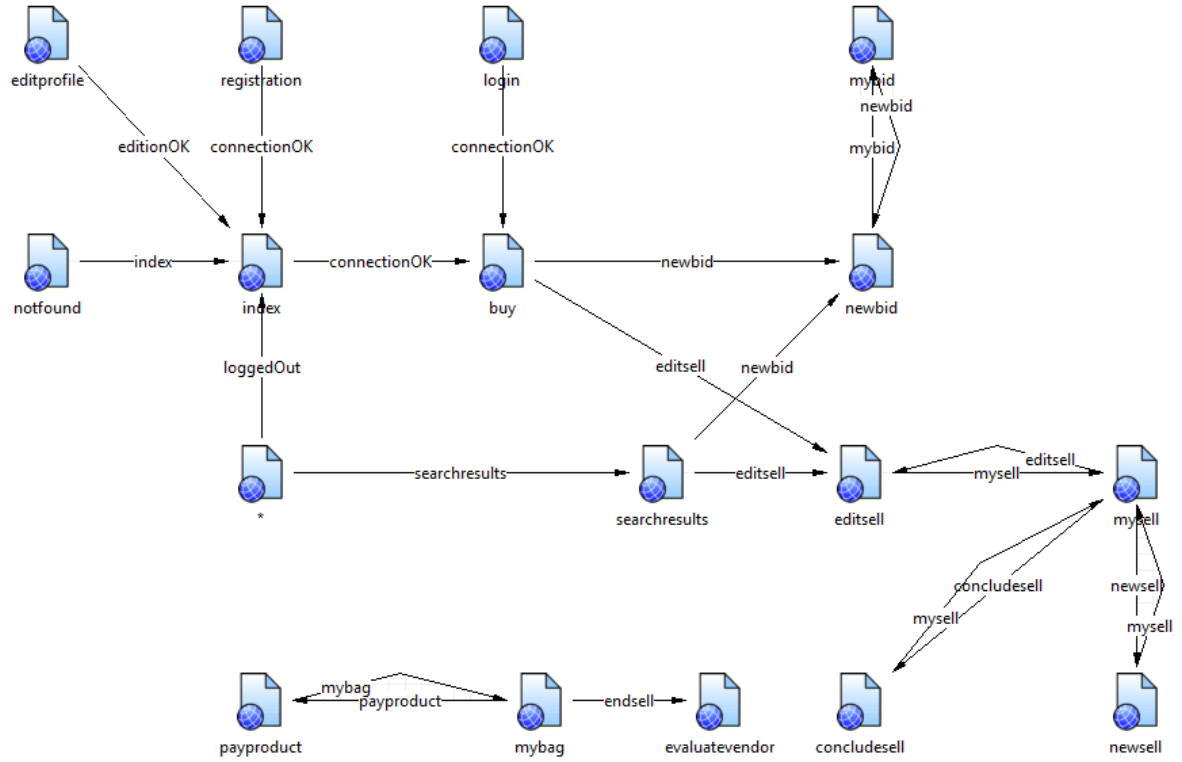

Fig. 6 – *faces-config.xml : règles de navigation*

A titre d'exemple, le schéma 6 nous présente l'ensemble des règles de navigation du site eBide.

5.2.2 Vue

Pages JSP : Notre site s'articule selon le schéma ci-après. On retrouvera en rouge les pages accessible uniquement à un utilisateur non identifié, en jaune les pages destinées à un utilisateur connecté et en bleu les pages accessibles à tous (quoique différentes selon l'état du visiteur). Ce diagramme est celui de notre site final, un certain nombre d'autres pages aurait été implémentées avec plus de temps (Entre autre, gestion plus poussées des paiements, messagerie interne, etc ...).

Nos pages .jsp ne contiennent en fait pas d'éléments de langage `JSP` à proprement parler. Tous ont été remplacés par les tags `JSF` adéquats.

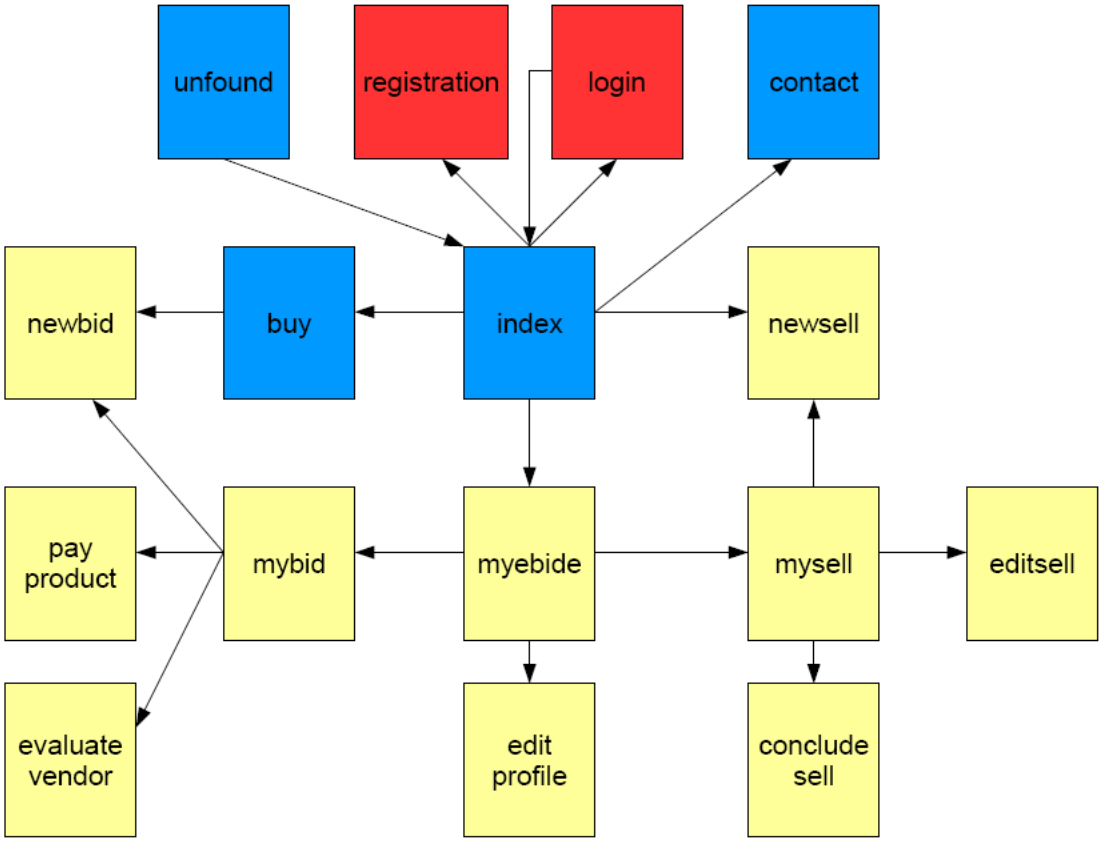

FIG. 7 – *Arcitecture du site*

Par exemple, pour afficher le nom de l'utilisateur actuellement logué, on se contentera de : `<h:outputText value="#{userController.user.login}"/>` Via faces-config.xml, on sait que **userController** fait référence à l'instance courante de USERCONTROLLER dans le scope session, et on peut accéder à ses attributs (les appels aux getters sont implicites).

Dans l'exemple ci dessus, le préfixe h : indique que l'ont fait appel à un tag de la librairie **html** de JSF (ici, on se contente d'afficher un texte. On pourra tout aussi bien générer des tableaux, formulaires, etc...).

On dispose également de la librairie **core** de JSF, préfixée par f : dans notre site. Par exemple :

```
<h:commandLink action="#{userController.actionLoadProfile}" value="...." >
  <f:param name="user" value="#{product.vendor.login}"/>
</h:commandLink>
```

Ici, on va associer le paramètre **user** avec la valeur définie à la requête résultant du commandLink. Ce paramètre sera récupéré au niveau du controleur.

Nous nous sommes appuyés sur la librairie OpenFaces, nottement pour l'ajout de composant Ajax. L'intégration s'est faite très naturellement : on dispose de tags spécifiques, préfixés par o :, qui sont intégrés directement dans la page. Par exemple : `<o:dateChooser id="endDate"value="#{userController.user.birthdate}"...` Cette ligne génère simplement un calendrier permettant de choisir une date (ici, la date de naissance lors de l'inscription).

Nous avons également utilisé une librairie additionnelle, `EL-Functors`, qui étend les possibilités des EL (Expression Language). A titre d'exemple : `<h:outputText value="#{bidController.actionGetBestBid$[product].amount}"/>` Le `$[...]` est spécifique à `EL-Functors`. Cela permet ici d'appeler la méthode `ActionGetBestBid` du BIDCONTROLLER avec passage de paramètres , ce qui est impossible de base (Seule des méthodes sans paramètres étaient appelables).

CSS : N'ayant pas de connaissances sur le CSS au début du projet, nous avons préféré ne pas partir d'un layout ou d'un template existant et de le faire à la main sans logiciel comme Dreamweaver afin de tester vraiment un maximum de comportements.

Dans un premier temps, en partant des pages statiques et en arrangeant leur structure en vue de leur appliquer du CSS, nous avons intégré les premiers composants comme l'entête, le menu, le bas de la page. Dans un second temps, lorsque nous avons commencé à traduire les pages statiques en `JSF` (sans réel contenu dynamique encore), nous avons du étudier les composants `JSF` et le code qu'ils génèraient afin de pouvoir voir comment leur intégrer du CSS. Pour appliquer du style à un composant `JSF`, on peut utiliser son attribut `styleClass` de la même manière que l'attribut `class` d'une balise HTML. Par la suite, lorsque le contenu des `JSF` est réellement devenu dynamique, nous avons pu tester une intégration pratique et intéressante du CSS dans `JSF`. Les composants qui permettent de gérer une suite de composants de manière cyclique comme panelGrid et dataTable ont de riches attributs relatifs au style des composants qu'il contiennent. Par exemple un composant panelGrid possède un attribut `columnClass` qui permet d'appliquer un style au colonnes du tableau. Si on lui applique deux classes : `columClass="classe1,classe2"` Alors le style classe1 sera appliqué aux colonnes impaires et le style classe2 aux colonnes paires. D'autres attributs facilitent l'application d'un style aux structures comme les tableaux.

Nous nous sommes heurtés à certains problèmes de positionnement auxquels nous avons tant bien que mal tenté de trouver une solution. Nous avons pu comprendre le modèle de boîtes de CSS (voir ci-dessous) et quelques astuces concernant le positionnement d'un bloc à l'intérieur d'un autre. Il y a certains comportements qui sont restées un peu plus mystérieux et cela vient du fait qu'en CSS, il y a un certain nombre de propriétés héritées et que chaque balise HTML a un comportement par défaut que l'on peut modifier (et notamment faire des choses pas très jolies pour s'arranger mais qui occasionnent des dommages collatéraux). L'autre inconvénient était que comme nous ne faisions que rajouter du code CSS à celui existant, certaines propriétés devenaient certainement inutiles mais il était difficile de revenir en arrière et de retester les effets de chaque propriété.

Le modèle de boîte de CSS est puissant mais son inconvénient est qu'on ne peut pas visualiser ces différents niveaux de boîte. Le moyen le plus visuel que nous avons eu pour « debugguer » les propriétés de positionnement lorsque celles-ci ne fonctionnaient pas comme nous le voulions était de rendre visible la bordure avec la propriété *border*. Elle permet de visualiser le bord de la bordure mais pas le bord de l'espacement (*padding*) ni de la marge (*margin*). Toutefois, visualiser l'espacement de deux blocs voisins en leur appliquant des *border* donne quelques indices sur la marge et permet de tester des

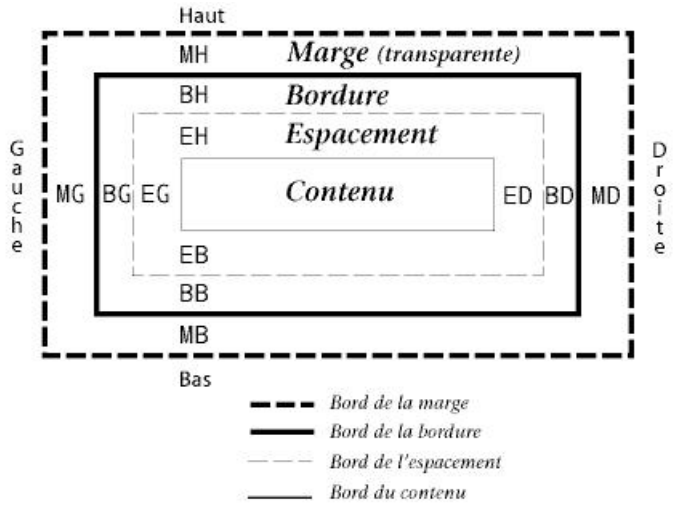

FIG. 8 – *Le modèle de boîte CSS*

propriétés dessus.

Finalement, il a semblé que nous parvenions petit à petit à quelque chose d'à peu près satisfaisant. Malheureusement, par manque d'expérience, nous n'avons pas effectué de test sur différents explorateurs et nous n'avons pas soumis le CSS à tout ce à quoi il aurait pu être sensible. Le CSS n'est pas une discipline des plus épanouissantes mais il a un potentiel énorme et permet de faire de jolies choses lorsqu'on comprend vraiment le fonctionnement des propriétés CSS et les effets de l'emboitage des balises sur ces propriétés.

Sur une grande majorité de sites commerciaux, on peut constater que les pages sont alignées à gauche. Cette contrainte aurait été plus simple à implanter que celle que nous avons choisie, à savoir celle d'une mise en page centrée.

6 Conclusion

Dans l'ensemble, nous avons trouvé ce projet très intéressant : outre le fait que la grande liberté laissée dans le choix du site était particulièrement motivante, ce fut l'occasion d'approfondir les notions vues en cours et TPs et de découvrir de nouvelles technologies par nous même. En cela, ce fut donc une expérience à la fois ludique et enrichissante. On regrettera simplement le couplage avec le projet de TDL, qui nous a obligé à limiter le temps que nous pouvions y consacrer.

A Référence

A.1 JBoss

Nous avons principalement utilisé la version 5.0.1 – GA de JBoss, disponible à cette adresse : `http://www.jboss.org/jbossas/downloads/`

A.2 JSF

L'implémentation choisie est celle fournie par JBoss, aucune librairie annexe n'est donc requise.

A.3 Librairies additionnelles

OpenFaces : Les librairies nécessaires pour utiliser OpenFaces sont disponibles à cette adresse : `http://openfaces.org/downloads/2.0/openfaces-2.0.eap1.zip`

EL Functors : disponible au téléchargement ici : `http://el-functors.sourceforge.net/`

B Note sur le fonctionnement du projet

Pour faire fonctionner le site, il faut préalablement initialiser la base de données. Pour cela, il suffit d'utiliser le script situé dans le dossier **ressources/init.sql** du backend.