Fil rouge : instructions enseignants
Module FIP INF 211

Présentation globale du fil rouge

Rappeler que l'objectif du FR est un objectif principalement techno. Le travail sera organisé en binômes.
Comme le sujet est nouveau et qu'on ne sait pas la quantité de travail personnel que le projet exigera, il leur est demandé de joindre une un fiche tâche/temps. Un exemple de avec des explications pour la remplir est disponible sur Moodle.

Ils disposent de trois documents imprimés qui sont aussi disponibles sur Moodle. Les deux premiers concernent toutes les séances du fil rouge, ils ont été distribués en début de module.

- Doc (court) de présentation du fil rouge.
  - Rappeler que le fil rouge couvre les trois principaux domaines du module : BD, archi technique et XML avec un focus plus important sur la partie archi. D'ailleurs (cela leur a déjà été dit), la partie archi du FR peut être vu comme un « gros » TP pour cet enseignement. En ce sens, les documents des séances sont relativement détaillés et guident le travail des élèves.
  - Le déroulement du projet a été modifié depuis l'impression des documents pour les élèves, la nouvelle version du document est dispo sur Moodle. La séance consacrée à la *gestion des services métier et de la couche présentation* se déroule maintenant sur deux séances. Cette séance supplémentaire se déroulera le 7/10. 4h30 au lieu de 10h30 pour cette partie soit un total de 13h30 encadrées au lieu des 10h30 prévues initialement. Le travail demandé reste le même.
- Le cahier des charges.
  Le sujet est le développement d'une application Web permettant la gestion d'un cabinet de recrutement.
  La partie principale de ce document est la description complète des fonctionnalités à implémenter. En complément du CdC, une démonstration de l'outil est disponible à l'URL :
  http://srv-labs-001.enst-bretagne.fr/CabinetRecrutement_WEB/template.jsp
  Peut-être une rapide démonstration ?
  Leur dire que chaque fonctionnalité est étiquetée avec un niveau d'importance : vitale et importante (obligatoire), mineure (optionnelle). On peut avoir une note correcte avec uniquement les fonctionnalités obligatoires mais l'implémentation des fonctionnalités optionnelles est un plus.

De façon plus annexe (documents non imprimés, disponibles uniquement via Moodle) :

- La fiche tâche/temps.
- Le document de Mayte sur l'utilisation de la machine virtuelle : *description de l'environnement de travail*.

Séance 1 : *conception des données de l'application*

Durée, 1h30 : pour les élèves, pas mal de choses à lire et à faire.

- Le document de la séance 1.
  Pas mal de blabla pour expliquer le contexte technique du fil rouge :
  - La machine virtuelle à utiliser : un document écrit par Mayte en explique la manipulation.
  - Au sein de la machine virtuelle, les outils qui seront utilisés : *Eclipse JEE*, *serveur d'applications GlassFish*, *SGBD PostgreSQL*, *pgAdmin* et/ou *Aqua Data Studio*.
  - Un rapide topo sur l'architecture de l'application. Note : cette séance se déroule en amont de la partie ARCHI, ils verront ça plus en détail plus tard mais ils ont ici quelques infos.
  - Un aussi rapide topo sur l'organisation des projets dans Eclipse.

  Le travail technique à réaliser est décrit à partir de la page 7 :
  - Dérivation du schéma conceptuel donné en un schéma logique (ils l'ont déjà fait en PC). Il n'y a pas d'héritage et globalement (il me semble…) pas d'ambigüité non plus.
  - Génération du script de création de la base de données pour PostgreSQL à partir du schéma logique.
  - Quelques éléments syntaxiques pour PostgreSQL sont fournis.

  Le doc se termine par les livrables à fournir avant la séance suivante.

Séance 2 : *gestion de la persistance des données (JPA)*

Durée, 3h00 : un peu plus de travail de travail technique que dans la séance 1.

Quatre parties dans le document (les deux dernières décrivent le travail à réaliser) :

1. Déroulement.
2. Infos préliminaires.
3. Génération des classes entités.
4. Création des DAO.

Si vous voulez être bavard, il est possible de situer le travail de la séance dans le schéma d'archi présenté dans le document de la séance 1, page 5. Cela situe le travail même si Mayte n'a pas encore abordé les archis n-tier avec eux. Il n'est bien évidement pas nécessaire d'entrer dans les détails.

Le domaine couvert par cette partie est similaire à celui du TP JPA. Attention l'approche est ici inverse de celle du TP : *bottom-up* (les classes sont générées à partir de la base) à la place de *top-down* (le contraire donc…). Leur préciser :

- Une courte explication a été faite dans le doc de la séance 1.
- Le travail est plus « facile » que le TP JPA : pour une bonne compréhension ils ont néanmoins besoin d'analyser la structure des classes ainsi que le mapping.

Partie *informations préliminaires*.

Dans cette partie, la présentation de la config d'*Eclipse -> PostgreSQL* et *Eclipse -> GlassFish* est donnée à titre indicatif. En revanche, pour la config de *GlassFish -> PostgreSQL*, il est important que ce soit compris (j'espère que le petit schéma aidera). L'application, une fois déployée, se sert de cette configuration pour se connecter à la base.

Partie *génération des classes entités*.

RAS, s'ils suivent correctement le déroulement (et que les tables existent dans leur base de données…), ils se retrouvent avec les classes entités et le mapping tout fait ! Si l'informatique était tout le temps comme ça, ils prendraient tous l'option info en 3A !!!

Leur préciser tout de même que le paramétrage choisi est un paramétrage par défaut qui posera ultérieurement des problèmes qu'ils ont commencé à aborder dans le TP JPA : les erreurs liées aux associations LAZY. Celles-ci ne doivent pas apparaître dans la séance. Au cas où, les faire régénérer temporairement les classes entités en choisissant dans

l'écran 3 la valeur *EAGER* dans le paramétrage *Association fetch*. La gestion de ces problèmes sera vu au cours des séances 3 et 4.

Partie *création des DAO*.

Le premier travail consiste à terminer le DAO existant : *EntrepriseDAO*. Les méthodes CRUD à écrire sont expliquées et ils disposent d'infos sur les méthodes de la classe *EntityManager* à utiliser. Ils disposent aussi d'une courte explication du langage de requêtes JPQL (*Java Persistence Query Language*).

Note : certains poseront peut-être la question de savoir pourquoi il n'y a pas ici de code de gestion des transactions comme dans le TP JPA (*tx.begin()*, *tx.commit()*, etc.). Par défaut (c'est le cas ici), lors de chaque appel d'une des méthodes du DAO, une transaction est créée et le code nécessaire encadre alors cet appel. Le *begin* et le *commit* existent mais sont gérés de manière transparente par le conteneur d'EJB. L'annotation suivante ajoutée à chaque méthode *@TransactionAttribute(TransactionAttributeType.REQUIRED)* matérialise le comportement par défaut. Elle n'est pas obligatoire et alourdit la lecture du code.

La suite du travail consiste à développer les autres DAO sur le modèle d'*EntrepriseDAO*. Certains de ceux-ci ont besoin de méthodes particulières que j'ai listées dans le document.

Partie *procédure de contrôle du bon fonctionnement des DAO*.

Pour faire plaisir à Mayte, ☺, on ne parlera pas ici de tests (dans le sens JUnit) mais de bon fonctionnement…

Contrairement au TP JPA, il n'est pas possible de faire une classe exécutable (avec un *main*) qui instancierait un DAO pour en appeler ensuite les méthodes. Un DAO connait son *EntityManager* grâce au mécanisme d'injection de dépendances géré au sein du conteneur d'EJB : c'est le rôle de l'annotation *@PersitenceContext* au-dessus de la déclaration de l'EntityManager (*EntityManager entityManager;*). Il en résulte que dans notre cas, pour appeler une méthode d'un DAO, ceci n'est possible que dans l'application déployée. C'est pourquoi les tests (!!!) se font au sein de la servlet *ControlesDAOServlet*. J'espère que les explications du document accompagnées d'un exemple est suffisant. Je peux passer dans les salles dans le cas contraire.

Pour les livrables, leur dire de ne pas oublier la fiche tâche/temps.

Séances 3 et 4 : *gestion des services métier et de la couche présentation*

Durée S3, 1h30, durée S4, 3h00.

Comme convenu avec les élèves, la séance 3 peut être vu comme la fin du travail commencé à la séance 2 (classes entité JPA / composants DAO) ou comme la séance du début du travail de cette étape. À priori, je pense que la majorité des élèves choisira de terminer la séance 2.

Sur Moodle, ils disposent pour cette étape de plusieurs choses :

1. Le document de séance : trois parties, voir plus bas.
2. Un document annexe présentant les composants applicatifs du corrigé. Le découpage en composants qui y est proposé n'est donné qu'à titre indicatif. Ils peuvent, ou non, le suivre. S'ils décident de choisir autre chose voir oralement avec eux leur justification et estimer s'ils ne font pas fausse route.
3. Un exemple de gestion de session en JSP pris du corrigé. Voir le doc. de séance.
4. Deux liens indiqués dans le doc. de séance vers des supports sur la technologie HTML et sur la technologie JSP.

Le document de séance est organisé en trois parties qui ne se dérouleront pas forcément de manière temporelle. Je leur recommande plutôt de faire plusieurs itérations. D'autre part, ce document est plutôt un document qui fournit de l'information plutôt qu'un document donnant des instructions.

Partie *poursuite du développement du composant ServiceEntreprise*

Comme pour les DAO, une ébauche de cette couche existe déjà avec la présence du composant EJB *ServiceEntreprise*. La première étape consiste à le compléter. L'essentiel du texte décrit l'état actuel du composant. La bonne compréhension de ce qui est déjà fait devrait permettre la poursuite du travail.

Partie *ajout de nouveaux composants*

L'essentiel de cette partie consiste à décrire l'utilisation de l'assistant de création d'un composant EJB (bean session). C'est queque chose qu'ils ont déjà fait avec les composants DAO. La seule différence ici est que les composants de la couche métier doivent avoir une interface distante.

L'autre sous-partie parle du référencement des nouveaux composants au sein de la classe *ServicesLocator*.

Partie *la couche présentation : l'interface graphique de l'application*

Dans cette partie sont données des informations sur la technologie JSP (un lien vers une présentation rapide de la techno est donné en début de document). Pour introduire la techno, je commente la page JSP *liste_entreprises.jsp* qui est représentative de l'essentiel des pages JSP qu'ils auront à fournir, au moins pour celles qui réalisent de l'affichage de données obtenues auprès de la couche métier. RAS sur ce contenu à priori.

Les points durs sur la partie IHM seront sans doute :

1. La gestion des formulaires Web. Si ce point vous pose un problème, je peux passer dans les salles.
2. La gestion des sessions HTTP. J'ai déposé sur Moodle un exemple d'une gestion de sessions tiré du corrigé. Là encore, si besoin, je peux passer dans les salles pour des explications.

Problème non référencé dans le document : exceptions sur les associations marquées à LAZY

Un problème qui sera certainement troublant pour eux, que je n'ai pas documenté et sur lequel il tomberont à coup sûr…

Dans la génération des classes entité, le choix qu'il leur était demandé de faire sur un paramètre appelé *Fetch type* était *default*. Ce choix se traduit par du *EAGER* sur les association de cardinalité 1 (pas de problème pour y accéder) mais *LAZY* sur les cardinalité multiples (erreurs à prévoir…).

L'erreur qui remonte avec *EclipseLink* (rappel, le provider JPA du projet fil rouge n'est pas *Hibernate* et l'erreur est donc différente) est de la forme :

```
Exception Description: An attempt was made to traverse a relationship using indirection that had a null Session.
This often occurs when an entity with an uninstantiated LAZY relationship is serialized and that lazy relationship
is traversed after serialization.  To avoid this issue, instantiate the LAZY relationship prior to serialization.
```

Pour gérer ça plusieurs solutions :

1. Régénérer les classes entité en choisissant *EAGER* pour le paramétrage du Fetch type. Dans ce cas, toutes les associations du mapping sont positionnées à *EAGER*. Plus aucun problème de *LAZY* mais leur faire pointer les difficultés potentielles : gestion de la mémoire (effet spaghettis quand on ramène une seule instance), perte de performance avec une multitude de requêtes SQL sur la base pour pouvoir tout ramener, etc. Note : même si ce n'est pas une bonne solution, c'est celle que j'ai choisi pour le corrigé (!). Sur de petites quantités de données, c'est gérable…
2. Gérer, par association, le choix EAGER. Note, Eclipse fournit un outil qui permet ce paramétrage (qui ne fait que modifier le mapping de l'association). Accès : menu *Window* > *Show View*. Choisir *JPA* puis *JPA Details*. Gestion du mapping du type presse-bouton, pas mal.
3. Ajouter un *join fetch* dans la requête JPQL qui ramènera les instances.
4. Ne pas utiliser les interfaces distantes. En effet quand une association est à LAZY si le consommateur du service et le service en lui-même sont dans le même conteneur, le contenu LAZY est renseigné dynamiquement et l'erreur LAZY disparait. Il faut dans ce cas dans le *ServicesLocator* récupérer les composant EJB par la classe et non par l'interface distante. Ce qui est possible ici et ce qui serait d'ailleurs le choix à préconiser.

Suivant le niveau de compréhension des élèves (+ faible vers - faible), on peut aller du choix 1 (voire 4) au choix 2 (voire 3).

Séance 5 : *gestion de la syndication des informations*

Durée 3h.

La durée de la séance est largement suffisante pour effectuer le travail demandé. Deux flux RSS sont demandés dans le cahier des charges : le flux permettant l'accès à la liste des offres d'emploi (initialement obligatoire, devenu optionnel) et celui donnant accès à la liste des candidatures (optionnel). Le second flux se gère exactement de la même manière que le premier sans difficulté technique additionnelle.

Le travail de cette séance est complètement satellite du travail principal du projet. L'objectif est double :

1. Montrer que les fonctionnalités métier développées précédemment sont « factorisables » et peuvent donc être utilisées dans un autre contexte. La liste des offres d'emploi sert dans l'interface Web de l'application, c'est la même fonctionnalité qui est utilisée par le gestionnaire de flux RSS. Ce gestionnaire aurait pu être déployé dans une application complètement indépendante. Pour des questions pratiques ce n'est toutefois pas le cas.
2. Le deuxième objectif est lui plus technologique : montrer la relative facilité de gestion de contenu XML à l'aide d'outils de plus haut niveau que les outils dont ils ont entendu parler en cours (parseurs XML : SAX, DOM, etc.).

Une petite réflexion pourrait être faite avec les groupes les plus avancés qui est liée à l'objectif 2 : de manière plus générale, il pourrait être intéressant de leur faire sentir que malgré la relative complexité de l'application qu'ils développent, le code qu'ils doivent produire est relativement concis.

Par exemple :

- Dans l'application ils ont développé un certain nombre de composants EJB qui peuvent être accédés à distance (RMI et/ou WebService), dans un contexte transactionnel, etc. Hors complexité algorithmique interne, quasiment inexistante ici, la gestion logistique de l'application est restreinte : quelques annotations tout au plus. Le reste est du code Java, simple, tout ce qui a de plus classique.
- La gestion d'un flux XML valide par rapport à un schéma n'est pas forcément une chose aisée à développer si on passe par des outils de plus bas niveau. Ici, après une génération automatique de classes, construire un flux RSS relativement complexe dans sa structure ne demande qu'une trentaine de lignes de code !

L'idée est de montrer qu'on est capable de faire des choses compliquées (*très*) facilement à condition d'avoir une maîtrise des environnements techniques, des frameworks, des API, etc. Plus que les avantages syntaxiques d'un langage par rapport à un autre (Python est plus concis que Java…), le plus important reste « l'écosystème » qui l'entoure. La complexité de l'application est autant, sinon plus, une complexité liée à l'environnement technique (serveur JEE) qu'au code de l'application.

<u>Déroulement de la séance</u>

Le travail est organisé en deux parties avec un support verbeux (un peu trop ?) avec lequel ils devraient être relativement autonomes :

1. Comprendre comment est structuré un flux RSS. Pour ça ils ont accès au schéma XML et à sa représentation graphique dans le document de séance. Ils peuvent aussi examiner le code source d'un des flux de l'application exemple disponible. Pas de complexité importante dans cette partie, il suffit de comprendre l'imbrication des éléments RSS.
2. Développer leur gestionnaire de flux RSS en se servant de la technologie JAXB. Le travail se résume à compléter deux méthodes (une par flux) dans une classe dont l'ébauche est fournie. Je vous fournis dans un document à part le code de cette classe qui fera office de correction.

Séance 6 : *recette de l'application*

Durée 1h30.

La fiche recette de l'application du fil rouge est construite à partir du cahier des charges et vise à vérifier si les fonctionnalités ont été implémentées ou non.

Remarques :

- Les fonctionnalités sont listées avec leur niveau d'importance. Rappel : vitale et importante = obligatoire, mineure = optionnelle.
- Une fonctionnalité peut être réalisée ou non, si elle l'est, la fiche demande si la prise en compte de certaines contraintes est réalisée.
- Il y a certainement des doublons : par exemple, la vérification de la fonctionnalité de connexion existe et en plus on demande de vérifier qu'une fonctionnalité est accessible à un compte particulier. On fait ce qu'on peut...
- Je n'ai pas mis dans la fiche la fonctionnalité d'indexation des offres et des candidatures, c'est en fait une sous-partie du référencement d'une offre ou d'une candidature. La correspondance offre(s)/candidature(s) fait l'objet d'une fonctionnalité présente dans la fiche.

Déroulement proposé

La fiche est donnée en début de séance, ils ont 10 minutes pour la remplir. Ensuite, à raison de 6 binômes par groupe et 10 minutes par binôme, il reste 10 minutes de rab... un peu court !

Modalités de correction

Ce que j'aimerais prendre en compte :

- la réalisation des fonctionnalités obligatoires.
- Un bonus pour l'investissement dans le travail (curiosité, intérêt, etc.)
- Le respect, dans l'implémentation de l'application, de l'architecture proposée.
  On peut leur demander oralement une explication sur la structure de leur code se basant sur une fonctionnalité particulière.

Exemples de notation

- Toutes les fonctionnalités obligatoires (et uniquement celles-ci) + investissement ultra minimal + respect très moyen de l'archi → 10.
- Uniquement les fonctionnalités obligatoires + bon investissement + quelques accrocs (traitements faits dans les JSP par exemple) → 12, 13, 14.
- Toutes les fonctionnalités + très bon investissement + respect de l'architecture → + de 16.

Mon échelle part ici de la note 10, bien évidemment on peut avoir des cas particuliers avec des notes inférieures. Par ailleurs, je pense que le projet est un excellent moyen de les faire progresser et mettre des notes un peu « serrées » présente un risque de démotivation pour les années à venir qui irait à l'encontre de ce qu'on essaie de faire... on peut en discuter.