Institut Paul Lambin

BAC 2 Informatique de gestion

AAM

Synthèse AAM

Auteurs :
Christopher Sacré

Professeur :
L. Leleux

7 juin 2017

Table des matières

1 Prise de Notes

1.1 Semaine 2

1.1.1 Introduction à l'architecture

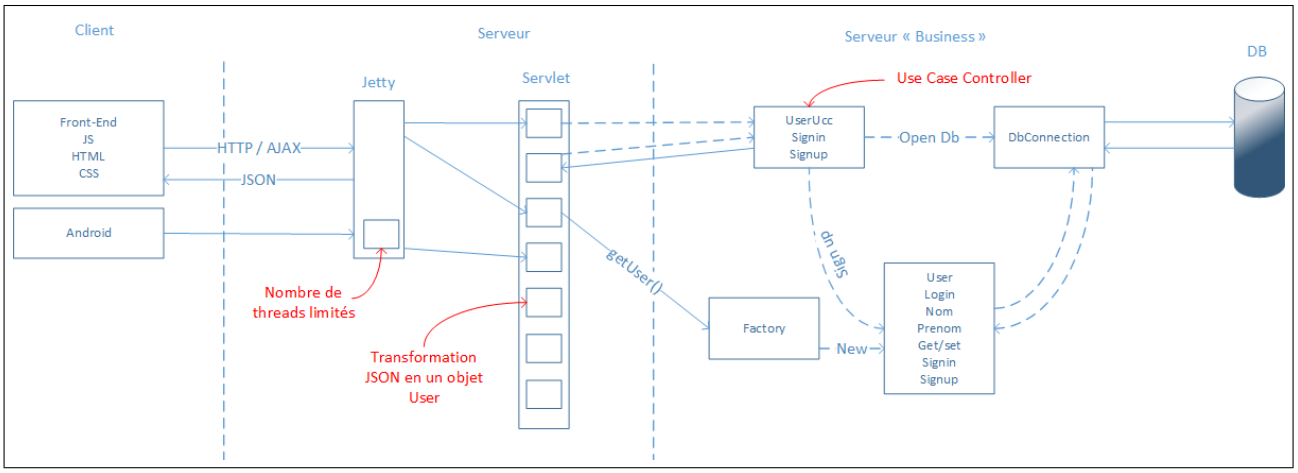


FIGURE 1 – Introduction du cours

1.1.2 Informations

Si vous souhaitez des informations supplémentaires, plusieurs logiciels ont été mentionné durant ce cours : Harmony, Visual Studio Code ainsi que Electron. (Il s'agit d'un cours nous introduisant les concepts de notre application ainsi que les bases de son architecture).

1.2 Semaine 3

1.2.1 Traitement d'un UseCase

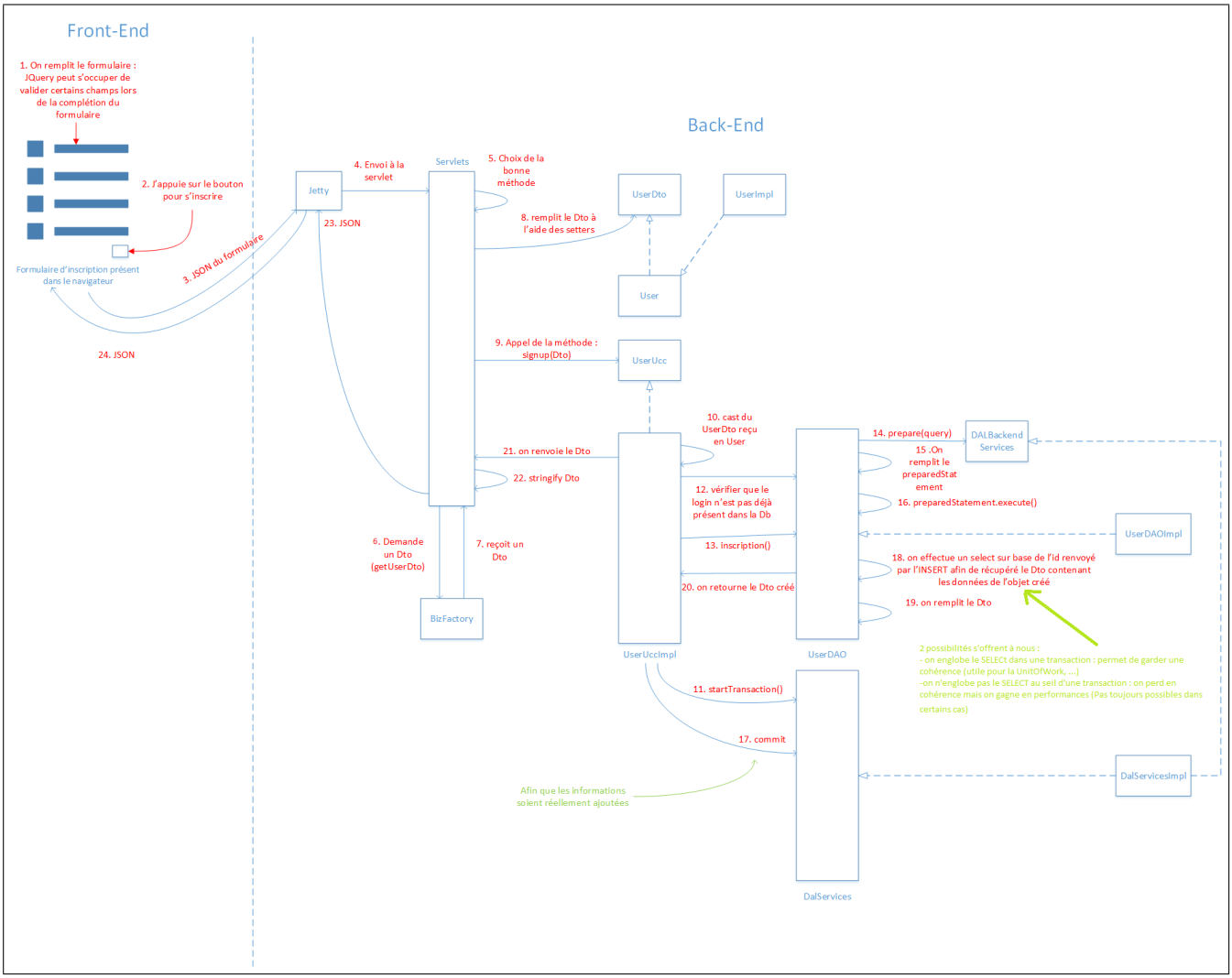


FIGURE 2 – Exemple d'utilisation de notre architecture (UC : s'inscrire)

1.3 Semaine 4

1.3.1 Architecture Monothread complète

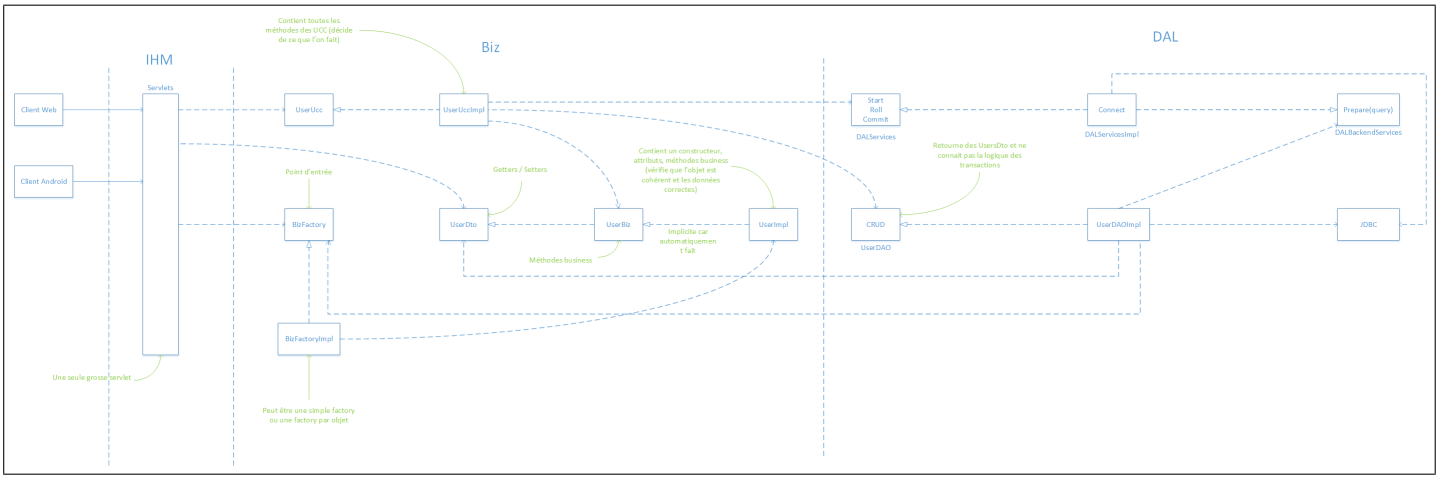


FIGURE 3 – Architecture Monothread complète

1.3.2 Informations complémentaires

Factoring : Il faut savoir que l'on mettra de nombreuses classes, ... public mais on tentera d'utiliser uniquement les points d'entrées (surtout au sein de la couche IHM), dans le même style, on utilisera de nombreuses interfaces afin de diminuer le nombre de dépendances concrètes et faciliter le changement entre l'environnement de prod et celui de dev. Par ailleurs, la servlet ne doit pas connaitre les méthodes bussiness (Car dans le cas contraire elle pourrait les modifier).

Traitement des INSERT : lorsque l'on insère, on tente de retourner l'entièreté de l'objet crée et non juste l'id, cela permet de vérifier que tout c'est bien passé.

Traitement des SELECT : point suivant est plutôt controversé et dépend de votre version des choses il s'agit de l'utilisation de transactions au niveau des select : on peut ne pas utiliser de transaction (perte de cohérence mais gain de performances) ou au contraire utiliser des transactions (perte de performances mais gain de cohérence).

Sécurité : Il est important que le back-end soit totalement indépendant du front-end (les données reçues ne sont en aucun cas sure).

Remarque : un framework nous pose des rails, cela nous permet uniquement de nous orienter , dès lors il faut parfois s'éloigner des rails.

1.4 Semaine 5

Main : Le main permet de démarrer Jetty, c'est lui qui s'occupe de créer les servlets Jetty. Il crée les Factory à l'ai de l'injection de dépendance. Dés que tout cela est créé, il se met en pause (Le main se lance au démarrage de l'application).

1.5 Semaine 6

1.5.1 ConnexionPool

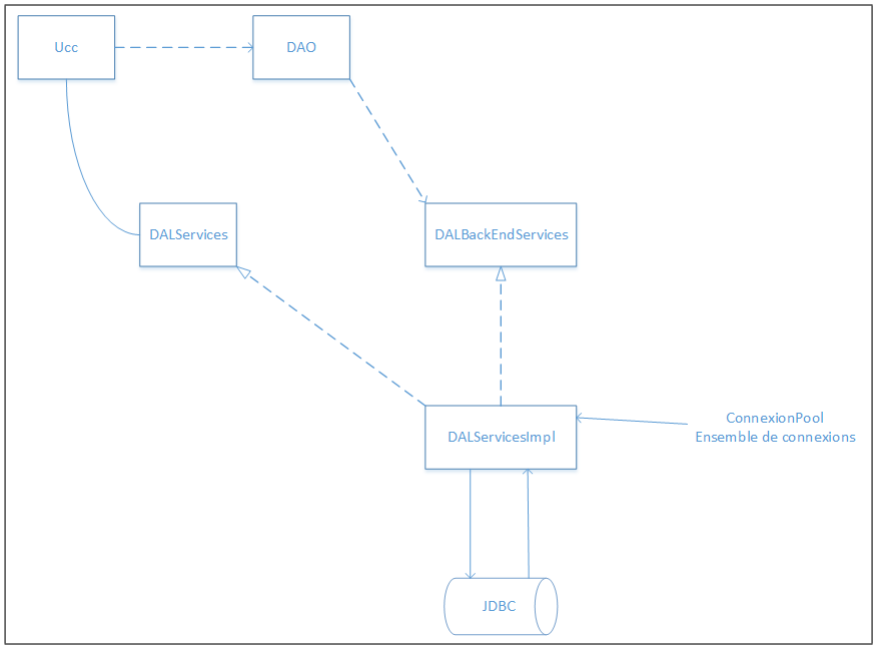


FIGURE 4 – Connexion Pool

Introduction : gros problème que l'on a tenté de résoudre est le fait que l'on ne pouvait ouvrir qu'une seule transaction à la fois (Il est totalement interdit d'ouvrir plusieurs transactions sur la même connexion en même temps, En cas de transactions imbriquées, si la première transaction à un soucis le rollback ne se propagera pas, mais s'arrêtera au premier commit).

Solution : Jetty à pour cela, un ensemble de Thread (Il n'en a pas un nombre infinis). On va donc remplacer notre précédente unique connexion par un ensemble de connexions (Le nombre de connexions à avoir est de notre ressort, et c'est donc à nous de faie ce choix).

Aide à la solution : On peut afin de nous aider à implémenter cela utiliser ThreadLocal (On aurait dès lors une connexion par thread, utilisation d 'une Map¡Thread, ConnexionDb¿ (pour cela on peut utiliser la méthode Thread.getId() qui renvoie l'id du threadCourant), ou plus facile encore ou pourrait utiliser la classe ThreadLocal, notre Map deviendrait alors un ThreadLocal¡Connexion¿, la clé est directement l'id du thread local (il s'agit tout simplement d'une map pour laquelle on a spécifié la syntaxe)).

Problème : le problème avec ces deux solutions vient du fait que si il y a une connexion par thread, on va donc multiplier le nombre de connexions (La taille du threadPool est un problème en soit (il faut pouvoir la fixer et cela dépend la plupart du temps fort de la couche Bussiness (Si il n'y a pas assez de Connexions, cela bloquera le thread tant qu'il n'y en aurait pas une de disponible).

Solution Finale : Afin de palier à tout cela, on va utiliser DBCP2 (Une librairie Java, DataBaseConnexionPool).

1.6 Semaine 7

1.6.1 Informations

Affichage des erreurs : Afin d'afficher des erreurs on peut utiliser System.err.println (La différence avec System.out.println est qu'il pourrait y avoir un décallqge au niveau de l'affichage).

Classe Logger : La classe Logger permet de définir une priorité au niveau des messages (Il plusieurs niveaux de priorité : SEVERE (valeur la plus élevée), WARNING, INFO, CONFIG, FINE, FINER, FINEST (valeur la moins élevée). L'avantage de cela est que l'on pourrait Logger sur un autre serveur , permettant ainsi d'empêcher la surcharge d'un appareil (Il est intéréssant de Logger : le temps de réponse, le type d'appareil utilisé , le type de route utilisé, ou tout ce qui vous semble intéressant d'être connus).

Profiler - SQL : permet de connaitre le temps d'exécution d'une réponse.

Sécurité : On pourrait rajouter une couche afin d'empêcher les attaques DDOS (Cette couche permettrait de filtrer les requêtes et de les dispatcher à des serveurs plus petit qui traiteraient alors les requêtes). Un autre point important pour la sécurité est qu'il ne faut jamais faire confiance à l'utilisateur, Il faut par ailleurs faire la distinction entre Admin et Utilisateur et utiliser JWT (cf cours de JavaScript).

1.7 Semaine 8

1.7.1 Optimistic Lock

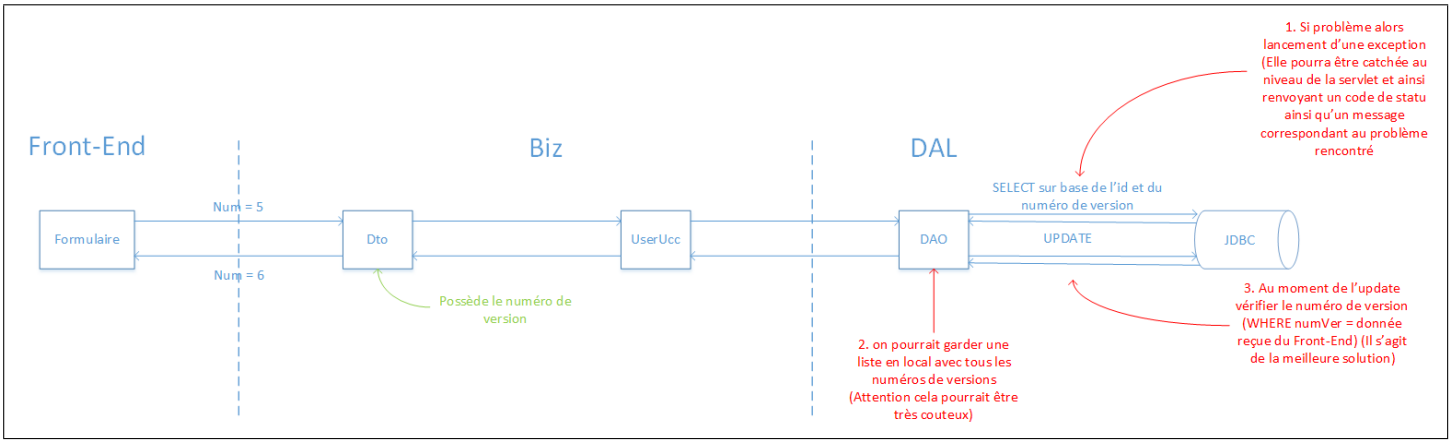


FIGURE 5 – Optimistic Lock

Introdution : un gros problème sur ce genre d'applications utilisée par plusieurs personnes est le problème de concurrence (Imaginons que deux personnes affichent la même donnée, le premier la modifie mais le premier qui l'avait également sous les yeux la modifie également, comment savoir quelles données sont correctes ?, comment faire pour que l'application reste cohérente à ce niveau là ?). Il existe deux solutions à cela : l'Optimistic Lock et le Pessimistic Lock (Durant notre cours nous nous intéresseront surtout à l'Optimistic Lock).

Optimistic Lock : On va utiliser un numéro de version que l'on va incrémenter à chaque modification (lorsqu'un utilisateur veut modifier une ressource, on récupérera l'id de l'objet ainsi que son numéro de version depuis le front-end et on le comparera avec ceux présent dans la base de données). On a décidé de ne pas travailler avec des timestamps car ils sont peu pratique à utiliser et qu'il peut y avoir un problème de décalage.

Amélioration possible : Refresh Intermittent , Une amélioration possible de notre système serait de prévenir le Front-End depuis le Back-End dès qu'une modification a lieu. Le problème vient du fait que normalement seul le client peut parler au serveur (C'est lui qui lance le processus de communication). Pour palier à cela il existe tout de même deux solutions : La solution Web-Socket (on ouvre un tuyau bi-directionnel entre les deux) ou la solution de Calling (on va faire des requêtes successives après un intervalle restreint afin de savoir si il y a eu une modification).

Amélioration possible : Exception , Une autre amélioration possible serait le lancement d'une exception si les numéros de versions ne correspondent pas, cette dernière serait récupérée au niveau de notre servlet qui pourrait envoyer une erreur correspondant au problème rencontré (ayant pour code d'erreur : 409).

Particularités DELETE : lors d'une suppression, on ne réalisera jamais de Hard Delete (DELETE en tant que tel sur la base de donnée, car cela pourrait avoir de grosses répercussions imaginons que l'objet supprimé soit référencé dans d'autres objets de la base de données, de plus des normes légales nous oblige à conserver nos données et on pourrait en avoir besoin pour faire des statistiques, ...) mais bien un soft Delete (On mettra juste un flag permettant de savoir si l'objet est supprimé à TRUE).

UnCaughtExceptionHandler : , La classe UnCaughtExceptionHandler est une classe que l'on peut instancier et assigner à un thread. Cette classe possède une fonction : uncaughtException qui permet si un thread a été stoppé par une exception qui n'a pas été attrapée, de fermer le serveur/ le thread qui l'exécute et d'ainsi permettre l'écriture d'un log afin d'enregistrer le problème.

1.8 Semaine 9

1.8.1 Introduction à la UnitOfWork

Problème rencontré : le problème de cette semaine concerne surtout le fait que chaque useCase de notre application nécessitait l'ouverture et la fermeture d'une transaction (dû au commit à chaque fin de UseCase). Cela cause de gros soucis de performances mais pire que cela i lfaut faire attention aux Usecases imbriqués (Il est totalement interdit de faire des startTransaction / commit imbriqués).

Solution : afin de palier à cela, on va utiliser un intermédiaire qui s'occupera de la logique de transactions (Il s'agira d'une classe à part qui s'occupera de la logique start-commit business). Cette classe s'occupera également de faire les appels aux DAOs, elle stockera les objets en vue d'UPDATE, INSERT et DELETE futur, au moment du commit à proprement parlé (lors de l'appel de la méthode commit de l'intermédiaire), l'intermédiaire va ouvrir la transaction, effectuer les opérations Db et commit. Attention, la UnitOfWork n'est pas nécéssaire pour le Multithreading, il s'agit surtout d'une amélioration pour les performances.

Stockage données : Etant donné que la classe intermédiaire garde en mémoire tout ce qu'elle doit faire on devra utiliser trois maps (une map pour INSERT, une map pour DELETE, une map pour UPDATE).

Améliorations au niveau des performances : Afin d'améliorer les performances on peut lors d'un INSERT et de plusieurs UPDATE sur un même objet, faire directement l'INSERT avec les bonnes données (Il faut faire attention au niveau de la cohérence), Pareil pour de multiples UPDATE, et dans le cas d'un INSERT ou d'un UPDATE suivis d'un DELETE, directement effectué le DELETE.

Solution Multithreading : Afin de permettre une solution multithreading on va devoir utiliser un ThreadLocal qui aura comme valeur la liste des trois Maps (Afin de permettre cela on va créer une classe interne (UnitOfWorkBag qui aura comme attributs les trois maps).

Solution start-commit imbriqués : On va utiliser une sémaphore (s'incrémentera de un à chaque start transaction de la classe intermédiaire, se diminuera de un à chaque rollBack, commit de la classe). Si la sémaphore est égale à 0 alors on lancera la procédure de commit autrement on effectuera aucun traitement.

1.9 Semaine 10

1.9.1 Impact de la UnitOfWork

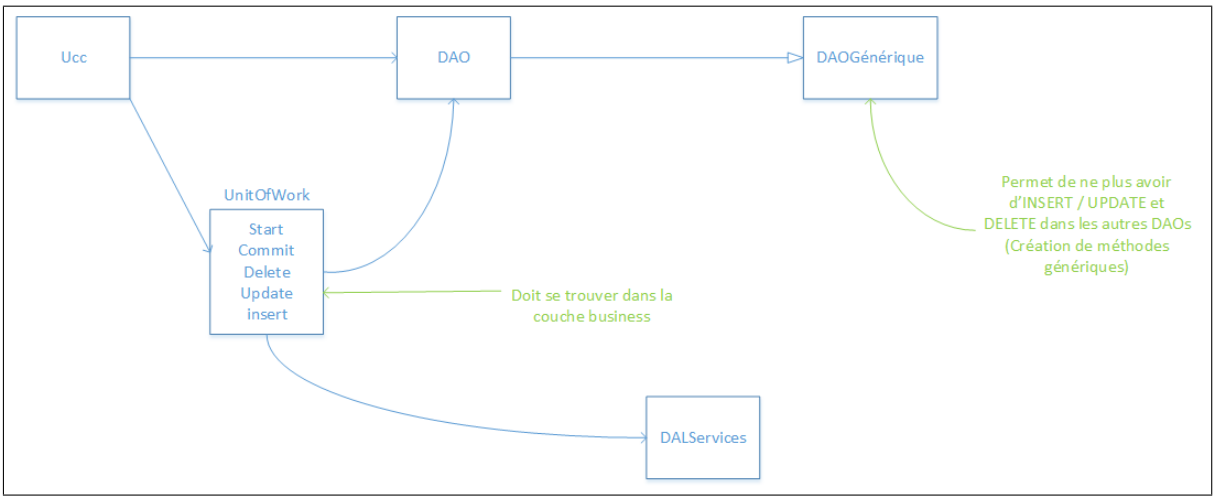


FIGURE 6 – Impact de la UnitOfWork

1.9.2 Améliorations de la UnitOfWork

Améliorations de la UnitOfWork

- *DAO générique :* on va créer un DAO générique qui s'occupera lui même de choisir la table de notre base de données à impacter en fonction de l'objet reçu. (Cette amélioration peut être remplacée par une interface DAO ayant trois méthodes : insert, update et delete ainsi tous nos DAO implémenteront cette interface et il nous suffira de choisir le DAO correspondant sur base de l'objet (afin de permettre cela (nous ne savon pas à l'avance le type d'objet reçu on utilisera l'introspection ainsi qu'un fichier properties (on gardera dans le fichier properties les DAOs responsable de chacun des objets (Afin de savoir de quel objet il s'agit on pourra utiliser la méthode objet.getClass()))).
- *Factory DAO :* Elle permettra de conserver la logique de recherche du DAO correspondant.
- *ORM :* ORM permet de mapper directement les objets java avec les objets en Db, pour cela on précisera sur chaque champs la colonne lui étant reliée (@Column) quant aux champs n'étant pas en rapport avec la db on le précisera à l'aide de l'annotation : @Transient.
- *JPA :* JPA permet d'être indépendant des bases de données.
- *Views* Views permet de faciliter l'injection de dépendances.

Différents type d'ids :

- *CID :* il s'agit de l'identifiant conceptuel (Conceptual ID). Il s'agit d'un identifiant proche du métier, il est unique au sein d'un type d'objets (PK de l'objet en Db). LE problème avec ce type d'id est que l'on peut avoir des objets de type différents ayant le même cid.
- *OID :* il s'agit de l'identifiant de l'objet (Object ID), ce dernier n'est pas utilisé par le client et est unique pour tous les types d'objets, aucun objet n'aura le même oid qu'un autre qu'ils soient de même type ou non à moins d'être égaux.
- *MID :* il s'agit de l'identifiant en mémoire (Memory ID). On ne peut donc en aucun cas se baser dessus.
- *UUID :* UUID est une classe java permettant de créer des ids de manière pseudo aléatoire (En utilisant le timestamp courant ainsi que les infos du pc (le nombre de cœurs, la fréquence CPU utilisée actuellement, ...).

Guice Guice est une libraire qui permet de faciliter le processus d'injection de dépendances (Pour plus d'informations à ce sujet c'est par ici.

1.10 Semaine 11

1.10.1 Les Sessions

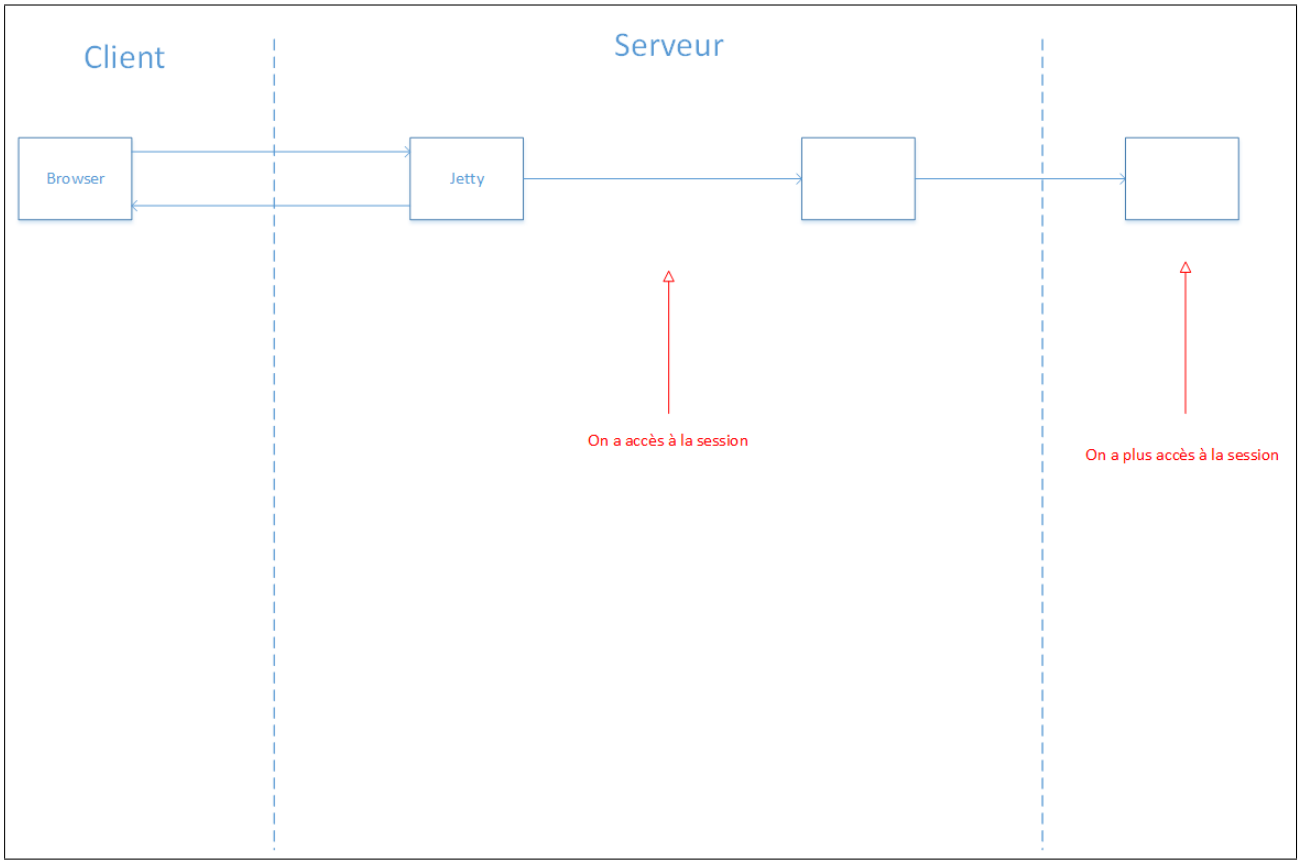


FIGURE 7 – Problème avec les sessions

Introduction : les sessions sont plutôt uniformes et communes à plusieurs langages. Elle permettent de garder les infos de la personne qui a effectué la requête (Le token JWT permet identifier l'utilisateur mais également de garder les informations pouvant être utile à chaque requête).

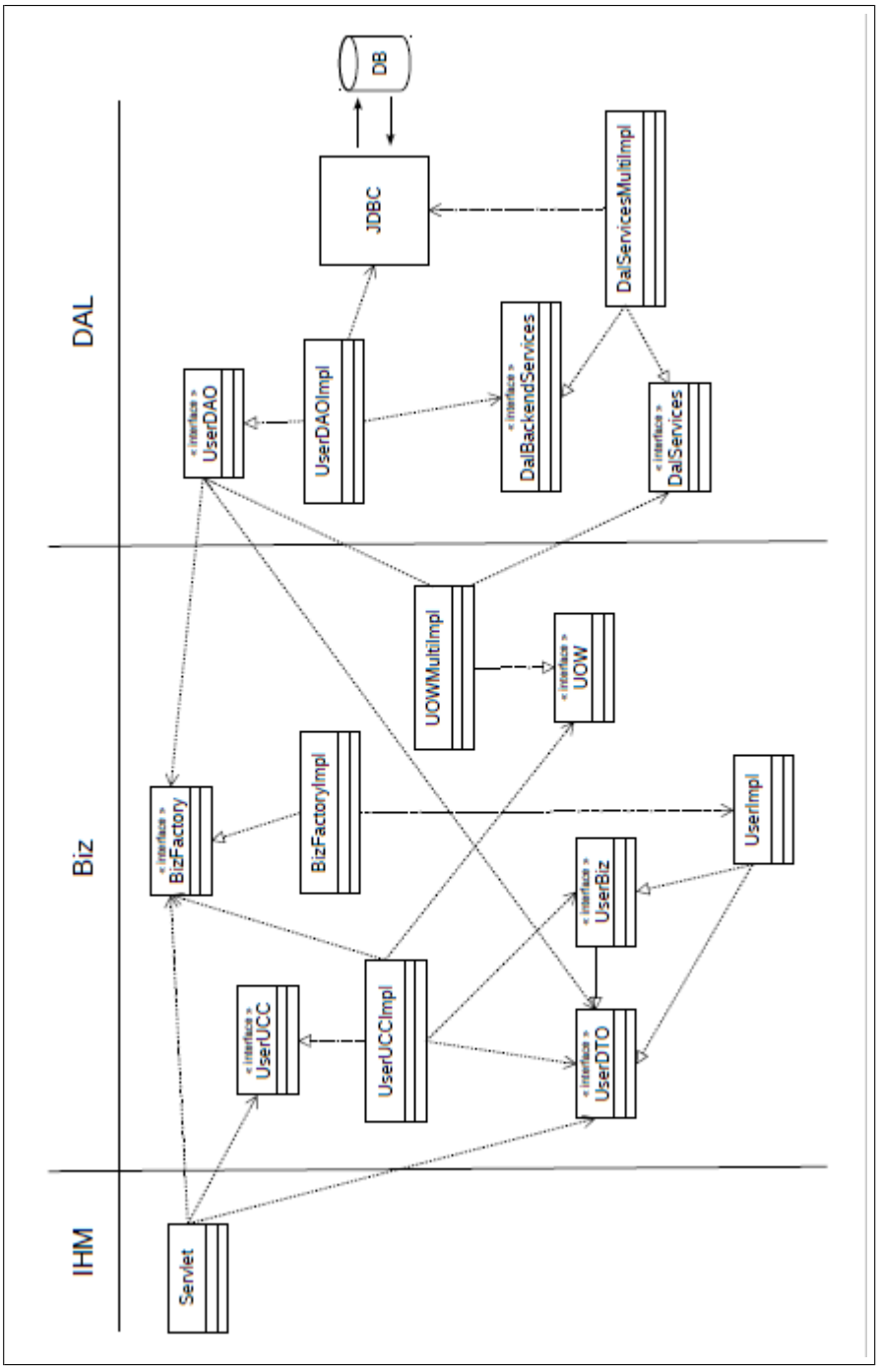
Problème rencontré : Comment va t'on faire pour avoir accès à cette session au niveau de la couche Business ? On ne peut pas la passer en paramètre (ce serait bien trop fastidieux), ni la mettre dans un fichier properties (cela ne peut se faire quant on à plusieurs utilisateurs).

Solution : Thread une solution au problème serait de stocker la session dans le thread. Afin d'avoir accès à cette session partout on va créer une classe SessionManager qui aura un attribut de type ThreadLocal (cet attribut pourrait même être statique, mais la dépendance concrète à peu d'importance).

Classe Config : on pourrait dès lors créer une classe Config disponible dans toute notre application qui contiendrait des infos devant être accesible un peu partout.

2 Analyse complète du système

2.1 Diagramme Complet


DB
JDBC
DalServicesMultiImpl
DAL
« interface »
UserDAO
UserDAOImpl
« interface »
DalBackendServices
« interface »
DalServices
UOWMultiImpl
« interface »
UOW
Biz
« interface »
BizFactory
BizFactoryImpl
UserImpl
« interface »
UserBiz
« interface »
UserUCC
UserUCCImpl
« interface »
UserDTO
IHM
Servlet

2.2 Description

Servlet : Permet de traiter les requêtes et de rediriger les requêtes vers les classes correspondantes au bon traitement (La plupart du temps passe le relais au UseCaseController correspondant).

BizFactory : Interface regroupant les méthodes publiques relatives à la logique de Factory. cette interface sert surtout à réduire les dépendances concrètes et ainsi favorise le passage entre l'environnement de production net celui de développement.

BizFactoryImpl : Classe implémentant BizFactory, cette dernière contiendra donc l'implémentation des différentes méthodes définies dans BizFactory, c'est donc elle qui contiendra réellement la logique de factory.

UserUcc : Interface regroupant les méthodes publiques relatives au UseCaseController d'un type d'objet. (Souvent ces UseCase seront des UseCase tels que ceux ci pourraient être définis dans un diagramme de cas d'utilisation, ils'agit en soit d'une tâche, d'un besoin de l'utilisateur).

UserUccImpl : Classe implémentant UserUcc, cette dernière contiendra donc l'implémentation des différentes méthodes définies dans UserUcc, c'est donc elle qui s'occupera réellement de traiter les cas d'utilisation relatif à un type d'objet.

UserDto : Interface ne contenant que les getters et setters d'un type d'objet (Il s'agit en soit d'un sac de données). Ils permettent de créer l'objet (via les setters) et de le rendre accesible dans la partie IHM de notre application.

UserBiz : Interface regroupant les méthodes Business nécessaire au sein des autres packaqges de notre application, cette interface sert surtout à réduire les dépendances concrètes et ainsi favorise le passage entre l'environnement de production net celui de développement.

UserImpl : Classe implémentant les interfaces UserDto et UserBiz. Étant donné que ces deux interfaces sont implémentées par une seule et même classe, passer d'une interface à l'autre est assez aisé.

UnitOfWork : Interface regroupant les méthodes relatives à la gestion de transactions business et donc il s'agit d'une interface définissant la méthode startTransaction, commit et rollBack business. (Attention cette classe n'est pas nécessaire pour avoir une application multithread fonctionnelle), cette interface sert surtout à réduire les dépendances concrètes et ainsi favorise le passage entre l'environnement de production et celui de développement.

UnitOfWorkImpl : Classe implémentant UnitOfWork, cette dernière contiendra donc l'implémentation des différentes méthodes définies dans UnitOfWork, c'est donc elle qui s'occupera réellement de la gestion des transactions business.

UserDAO : Interface regroupant les opérations SELECT, INSERT, UPDATE et DELETE relatives à un type d'objet. Cette interface sert surtout à réduire les dépendances concrètes et ainsi favorise le passage entre l'environnement de production et celui de développement.

UserDAOImpl : Classe implémentant UserDAO, cette dernière contiendra donc l'implémentation des différentes méthodes définies dans USERDAO, c'est donc elle qui s'occupera réellement des opérations d'INSERT, UPDATE, DELETE et SELECT d'un type d'objet.

DalServices : Interface s'occupant de la gestion des transactions , c'est cette dernière qui s'occupera de gérer les startTransaction, rollBack et commit et donc des transactions.

DalBackEndServices : Interface permettant surtout de garder la préparation des statement uniquement à l'intérieur du package DAL en le mettant à part des autres méthodes relatives au transactions. Cette interface s'occupera essentiellement de la préparation des statement.

DalServicesMultiImpl : Classe implémentant DalServices et DalBackEndServices, étant donné que cette classe implémente les deux interfaces, C'est cette dernière qui s'occupera à la fois de la gestion des transactions et de la préparation des statements. LA séparation en deux interfaces bien distinctes vient surtout du fait que l'on veut séparer ces deux logiques et empêcher la préparation des statements d'être accesible à l'extérieur du package DAL.