Projet multi-modules – Gettings Things Done
Rapport sur la persistance

Jérémy Braud Nicolas Bremard Boris Lucas
Cédric Krommenhoek

3 février 2010

Résumé

La méthode GTD est une démarche d'organisation personnelle applicable par chacun à l'ensemble de ses activités, tant professionnelles que privées. Décrite par David Allen dans son livre, la méthode GTD a pour objectif d'identifier avec sûreté ses priorité à tout moment, et d'agir immédiatement sur la priorité choisie. En effet, pour bien choisir sa priorité et s'y consacrer pleinement, il faut être certain de s'appuyer sur un système que l'on juge fiable.

Table des matières

Introduction

Dans le cadre de notre projet multi-modules, ayant pour sujet la conception et la réalisation d'un outil client de gestion de tâches suivant la méthode GTD [1], nous avons été confronté à des problématiques telles que l'analyse des besoins, les communications distances, ou encore l'ergonomie de notre interface homme-machine.

Ce rapport concerne quand à lui la problématique de persistance des données côté client.

1. Getting Things Done

Chapitre 1

Motivations

L'outil que nous développons requiert le stockage d'un grand nombre de données, comme par exemple les listes de contextes, de contacts, de liens des tâches, ou plus simplement les tâches elles-mêmes. Bien qu'il soit prévu que ces données soient sauvegardées sur un serveur distant, il est intéressant de pouvoir les conserver en local. L'outil devient alors exploitable quelles que soient les contraintes liées au réseau : le temps de réponse qui peut être élevé, ou pire, le serveur indisponible. La persistance des données permet ainsi de gagner en performance, mais aussi en portabilité. Le serveur ne sert alors plus qu'à synchroniser ponctuellement les données, afin de pouvoir y accéder depuis un autre ordinateur ou d'éviter la perte de données due à un problème survenant sur l'ordinateur du client.

Chapitre 2

Choix architecturaux

Avant de suivre le module d'objets distribués, les seuls moyens que nous connaissions pour conserver les données étaient soit d'accéder à une base de données distante, soit de sauvegarder les données sérialisées dans des fichiers XML. La première solution étant inenvisageable et la seconde très coûteuse et fastidieuse, nous avons bien évidemment opté pour la technologie que nous venions d'étudier : les entités JPA [1]. Ces entités sont gérées à l'aide du gestionnaire de persistance d'Hibernate [2], et nécessitent un moteur de base de données afin qu'Hibernate puisse les rendre persistantes. Sur conseils de M. Thimel, nous avons choisi le moteur H2 [3], une base de données embarquée très légère qui s'est avérée très efficace.

Il existe deux manières de spécifier les entités JPA : soit à l'aide de fichiers de description XML, soit en plaçant des annotations sur les classes représentant ces entités. Bien que la première solution présente l'avantage de ne pas modifier le code source, nous avons préféré choisir la seconde pour sa lisibilité et sa maintenabilité.

1. Java Persistence API
2. Hibernate – `https://www.hibernate.org/`
3. H2 – `http://www.h2database.com/`

Chapitre 3

Implémentation

Sommaire

3.1 Configuration des outils de persistance

L'unité de persistance d'Hibernate est définie par un fichier de configuration "persistence.xml". Ce fichier décrit les propriétés telles que le nom de l'unité, son emplacement, son pilote, etc. C'est également au sein de ce fichier que nous spécifions que nous désirons compléter, et non pas remplacer, la base de données lorsque nous redémarrons l'outil.

Nous avons rencontrés des difficultés à configurer H2 pour Hibernate, car la dernière version d'Hibernate présente des bugs, et les versions trop anciennes ont un dialecte incompatible avec celui de H2. Nous avons tout de même réussi à trouver des librairies compatibles afin que cela fonctionne.

3.2 Création des entités

3.2.1 Identification des entités

Pour commencer, nous devons identifier toutes les entités qu'il nous faut conserver localement. En plus des données du **compte** de l'utilisateur, nous avons besoin de :

- **tâches**,
- **projets**,
- **choses à faire**.

Les **contextes**, **contacts** et **tags** sont des objets communs entre différentes instances de projets ou tâches, et sont en conséquent eux-aussi des entités à persister. Les tâches et les projets peuvent disposer d'un **échéancier** leur étant propre, et cet objet étant complexe nous devons également en faire une entité. Il est possible d'associer un ou plusieurs **liens** à une tâche, mais comme le lien peut être représenté par une simple chaîne de caractères et qu'il n'est pas partagé entre plusieurs objets (contrairement aux tags ou aux contextes), il n'est pas nécessaire d'en faire une entité. Afin de correspondre aux interfaces du serveur distant, nous avons également ajouté la notion de **participant** à une tâche ou un projet, bien que cette information soit inutile dans notre conception.

Pour que nos entités aient toutes certaines propriétés, nous les faisons toutes hériter d'une classe abstraite qui ne sera pas persistée. Nous pouvons ainsi définir que toutes les entités ont un identifiant auquel nous pouvons accéder par la méthode "`getId()`", que toutes ces entités implémentent l'interface "`Serializable`" (pour une éventuelle communication distante), etc.

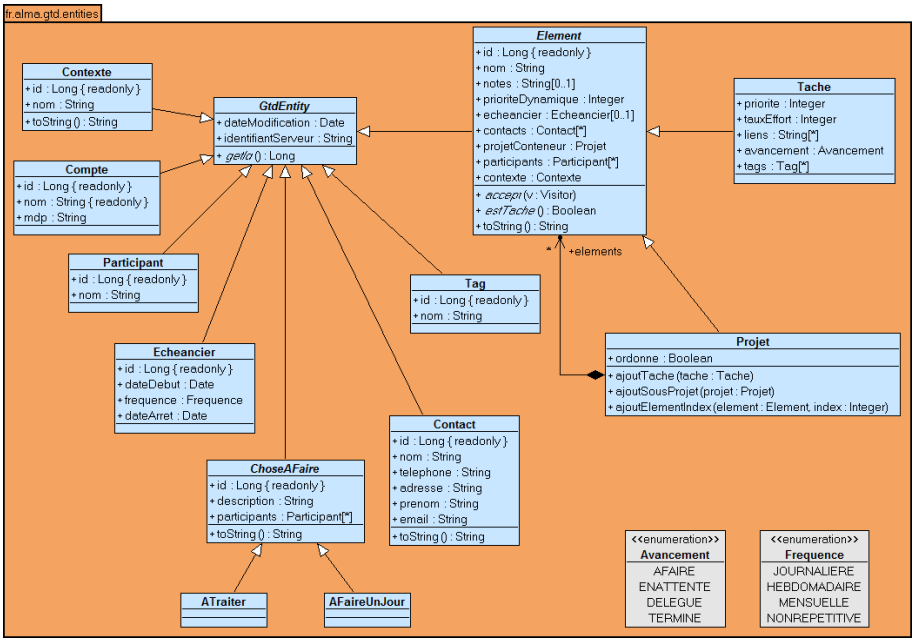

FIGURE 3.1 – Diagramme de classes des entités

Nous obtenons alors le diagramme de classes de la figure 3.1.

3.2.2 Stratégies de persistance employées

Relations entre les entités

La plupart des relations multi-évaluées entre entités sont des collections d'entités, que l'on peut spécifier sans problème au gestionnaire de persistance, à l'exception des collections de liens que nous avons cité précédemment. Pour la persistance de ces liens, nous avons employé l'annotation "`@CollectionOfElements`" qui ne provient non pas de JPA, mais du mécanisme de persistance d'Hibernate. Toutes les relations entre entités sont ici de type "`@OneToOne`", "`@ManyToOne`" ou "`@ManyToMany`", à l'exception de la relation entre un projet et les éléments qu'il contient. Dans ce cas précis, nous assurons la bidirection de l'association en spécifiant à l'annotation "`@OneToMany`" quel est l'attribut qui correspond dans la classe opposée. Nous obtenons le code suivant :

- dans "element.java" :

```
@ManyToOne
protected Projet projetConteneur;
```

- dans "projet.java" :

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
     mappedBy = "projetConteneur")
private Collection<Element> elements;
```

Préciser ce "mapping" nous permet également d'éviter la création d'une table de correspondance entre les projets et leurs éléments, ce qui optimise la base de données.

Arbres d'héritages

Les tâches et les projets sont agencés selon le patron de conception composite ; leur superclasse abstraite devient alors une entité supplémentaire. Nous avons choisi de représenter cet arbre d'héritage en utilisant la stratégie "une table par classe fille"[1], qui permet l'accès rapides aux entités projets ou tâches, tout en conservant un accès direct aux propriétés communes entres ces deux types d'entités. Nous avions pensé à spécialiser les projets en des projets ordonnés et non-ordonnés, mais nous n'avons jamais besoin d'accéder à seulement une des ces deux spécialisations, et la différenciation à partir d'un attribut booléen paraît alors plus adaptée.

En revanche, nous avons choisi de spécialiser les choses à faire en deux sous-classes, car nous n'accédons systématiquement qu'à un seul type de choses à faire. Nous utilisons dans ce cas la stratégie "une table pour une hiérarchie de classes"[2], car il n'y a pas d'attributs supplémentaires dans ces

1. annotation "`@Inheritance(strategy = InheritanceType.JOINED)`"
2. annotation "`@Inheritance(strategy = InheritanceType.SINGLE_TABLE)`"

spécialisations. Nous devons alors spécifier une colonne discrimante à l'aide de l'annotation "`@DiscriminatorColumn`" (dont nous pouvons déclarer le type [3]) pour préciser la spécialisation de l'objet persisté. Ici, les classes de choses à traiter et de choses à faire un jour sont différenciées respectivement par les annotations "`@DiscriminatorValue("A_TRAITER")`" et "`@DiscriminatorValue("A_FAIRE")`".

Enumérations

Les tâches ont quand à elles quatre états possibles, à savoir les tâches à faire, les tâches en attente, les tâches déléguées et les tâches terminées. Nous avons représenté ces états non pas par des spécialisations de la tâches, ce qui n'apporterait rien du point de vue conceptuel et qui nécessiterait la création d'un nouvel arbre d'héritage, mais par une énumération, ce qui est très bien reconnu par le gestionnaire de persistance. Même si les requêtes sur un type de tâches précis requiert une clause "`WHERE`", la plupart des requêtes concerne toutes les tâches à la fois. De la même manière, l'échéancier dispose d'une fréquence de répétition également représentée à l'aide d'une énumération [4]. Nous précisons au gestionnaire que ces deux énumérations sont enregistrées en tant que chaînes de caractères [5].

Gestion des dates

Afin de déclarer les dates présentes dans l'échéancier, nous avons employé l'annotation "`@Temporal`", qui permet, en plus de signaler qu'il s'agit d'un attribut de date, de spécifier que le type utilisé est "`TemporalType.DATE`". Les dates sont ainsi correctement persistées dans la base de données.

3.3 Création des DAO

Afin de faciliter la gestion des entités nouvellement créées, nous utilisons le patron de conception DAO [6]. Ce patron fait la séparation entre le code spécialisé lié aux requêtes faites sur les entités et le code métier. Pour développer ce patron, nous nous sommes grandement inspiré de l'exemple de code fourni par M. Thimel, que nous avons ensuite modifié et complété.

3. par exemple le type "`DiscriminatorType.STRING`"
4. annotation "`@Enumerated`"
5. type "`EnumType.STRING`"
6. Data Access Object

Chapitre 4

Conclusion

Ce projet nous aura permis de mieux comprendre le fonctionnement des mécanismes de persistance tels que le gestionnaire d'entités d'Hibernate, ainsi que de nous familiariser avec les entités JPA, autrement qu'avec les EJB.

Nous avons également découvert H2, qui est un formidable outil que nous réutiliserons certainement pour d'autres projets (comme par exemple le projet de fin d'études).