valtech days

Sami Jaber
Valtech Training

Implémenter une couche de persistance avec .Net

février 06

Résumé

Développer une couche de persistance est assurément une des tâches les plus difficiles lors de la réalisation d'une application de gestion. Alors que les techniques de mapping objet / relationnel sont restées pendant plus de dix ans l'apanage d'experts, de plus en plus de projets franchissent le pas. Avec cet engouement, ces mêmes projets payent souvent les pots cassés d'une mauvaise conception ou d'une méconnaissance des principes du mapping, dont la persistance est tout sauf transparente (malgré ce qu'affirment les présentations marketing).

Quels sont les pièges à éviter lorsqu'on doit faire du mapping O/R ? Y-a-t-il des modèles de conception à suivre ? La génération de code peut-elle apporter une solution alternative moins coûteuse et mieux maîtrisée ? Faut-il réserver cette technique à des petits projets ou des gros projets ?

Cette session s'attachera à décrire les grands principes du mapping objet / relationnel tout en décrivant au travers d'une étude de cas pratique les effets pervers d'une mauvaise conception. Les outils de persistance utilisés seront ceux du marché, qu'il soient propriétaires ou Open Source (nhibernate, Code-Smith, …). Quant à l'approche "techniquement agnostique", elle conviendra aussi bien aux aficionados de .NET que de Java EE.

valtech days

Table des matières

1. LE MAPPING O/R, POUR QUOI FAIRE ? 3
2. QUELLE DÉMARCHE ? 3
3. LES OUTILS ET STANDARDS 4
3.1 J2EE 4
3.2 .NET 4
3.2.1 Nhibernate 5
4. LES DIFFÉRENTES APPROCHES DU MAPPING 5
4.1 La persistance manuelle (ADO.NET, DataSet, …) 5
4.2 La génération de code (CodeSmith, NetTiers, etc …) 5
4.2.1 La persistance automatique 6
5. OPTIMISER SA COUCHE DE PERSISTANCE 7
5.1 Quelles sont les solutions ? 8
6. CONCLUSION 9

1. Le mapping O/R, pour quoi faire ?

Nombre d'applications orientée objet comprennent un accès à une quelconque base relationnelle. Jusqu'à présent, pour peu que le développeur ait pris soin d'adopter une conception purement objet, une phase de conversion s'avère indispensable. Cette phase de conversion encore appelée mapping objet/relationnel consiste à adapter le modèle relationnel à un modèle objet, navigationnel et hiérarchique.
Alors qu'un schéma purement relationnel représente les liens via des mécanismes de clés étrangères, un modèle objet s'appuie sur des dépendances fortement typées (Collections, Classes, ..). Les caractéristiques intrinsèques du développement objets tels que l'héritage ou le polymorphisme s'accommodent difficilement d'une vue relationnelle.

Ce problème a conduit au fil des ans les développeurs à effectuer d'incessantes pirouettes techniques pour adapter ces deux modèles par l'utilisation de composants techniques spécifiques (Pattern DAO pour Data Access Objects).
Si pour certains, la solution passe par l'acquisition d'une base de données nativement objet, pour d'autres au contraire, le problème est d'ordre applicatif. Les bases de données relationnelles ayant en effet prouvé leur maturité et leur capacité à monter en charge.

2. Quelle démarche ?

Que ce soit dans le monde J2EE ou .NET, il existe de nombreux motifs de conception ou design pattern permettant d'implémenter et d'optimiser manuellement (c'est-à-dire sans outil du marché) une couche d'accès aux données. Dans cette optique, une architecture multi-couches s'avère souvent bénéfique car elle tend à reléguer les phases de conversion dans cette couche spécifique appelée encore DAL (Data Access Layer) ou couche de persistance. Prendre en charge manuellement le mapping O/R nécessite de la rigueur de la part du concepteur. Car si cette démarche a l'avantage de tirer profit des compétences relationnelles existantes (SQL, procédures stockées, batch), elle devient vite fastidieuse sans un outillage minimum. En .NET, cela consiste à faire appel aux API ADO.NET afin d'extraire les données via des requêtes SQL pour les restituer sous la forme de graphes d'objets.

Plus le schéma relationnel est riche et complexe, plus cette tâche sera fastidieuse et répétitive. Sans compter les nombreux cas spécifiques tels que le chargement à la demande (Lazy Loading), la gestion du cache objet ou la prise en compte de l'héritage. Quelle soit automatique ou manuelle, la persistance requiert la mise en place d'un modèle d'objets métier ou modèle du domaine. Ce modèle contient généralement l'ensemble des classes métier d'une application indépendamment de leur structure de stockage (fichiers, sgbd, etc…). Il constitue le pendant des tables relationnelles dans le monde objet.

Nombreux sont les outils à proposer dans une approche bottom-up des générateurs de code à partir de schémas relationnels. A l'inverse, une fonctionnalité incontournable dans le monde du mapping est la génération de schéma et de scripts SQL à partir d'un modèle objet (approche top-down).

3. Les outils et standards

Depuis quelques années, le monde JEE croule sous les standards en tous genres, une situation différente côté .NET. Voyons ce que proposent les deux clans.

3.1 J2EE

La première spécification visant le mapping O/R dans le monde JEE fût l'initiative JDO (Java Data Object). Spécification adoptée massivement dans un premier temps par la communauté Java puis délaissée par les acteurs majeurs de l'industrie (IBM, Oracle, BEA) au profit de la norme EJB 3. Pour ses détracteurs, JDO serait trop structurant et pas assez riche d'un point de vue technique. En réalité, se cache derrière une bataille technologique et des enjeux financiers considérables. Contrairement aux EJB, JDO n'a pas spécialement vocation à s'insérer dans un serveur d'application, seul l'outil de persistance suffit à prendre en charge les différents services techniques. L'offre d'outils JDO étant déjà assez abondante et la plupart des éditeurs précédents ne fournissant aucun outil de ce type, le marché des serveurs d'application aurait subit le contrecoup d'une concurrence redoutable. Pour IBM " *Cette spécification se chevauche avec d'autres JSR en cours de développement (ndrl EJB 3). Dans un contexte de rationalisation et de simplification des outils JEE, il n'est pas souhaitable de voir se multiplier plusieurs modèles de développement concourrant aux mêmes objectifs* ». Ce qui explique le refus du géant de voter la dernière spécification JDO 2.0, l'une des plus aptes à supplanter les EJB Entités.

En marge de ces travaux, plusieurs acteurs réunis autour d'un projet OpenSource très convoité répondant au nom d'Hibernate emboîtent le pas aux géants de l'industrie. Aux côtés de Sun, Gavin King (l'auteur d'Hibernate), engagé aujourd'hui aux côtés de la société JBoss, rédige les prémices de la spécification EJB 3. En quelques mois, Hibernate gagne son pari et peut se targuer de réunir toute la communauté Java autour du projet EJB 3 (Enterprise Java Bean).

Autre signe des temps, Oracle, longtemps resté à l'écart de ce marché du mapping, réussit un tour de force en rachetant pour une poignée de dollars l'éditeur Toplink, grand spécialiste du domaine.

Et comme pour conjurer un sort déjà bien scellé, Oracle s'associe au groupe d'expert de la JCP EJB 3.

3.2 .NET

Côté .NET, la situation est sensiblement différente. Si l'éditeur de Redmond n'a jamais caché son apathie pour le mapping objet/relationnel, privilégiant une approche plus spécifique de l'accès aux données avec sa solution de stockage Sql Server, il semble que les choses évoluent différemment depuis quelques mois. En effet, la stratégie de Microsoft se redessine progressivement autour du futur langage C# V3 et du Framework Linq (Language Integrated Query), un langage supportant nativement les concepts de mapping avec plus spécifiquement Dlinq pour l'aspect mapping objet/relationnel.

Si pour l'heure Microsoft ne brille pas par son offre, la plateforme .NET peut tout de même se targuer de disposer de plus d'une cinquantaine d'outils (tous recensés sur le site sharptoolbox.com). Ils se nomment nhibernate, DTM (Evaluant), Pragmatier ou encore EntityBroker.

3.2.1 Nhibernate

NHibernate est le portage de l'excellent Hibernate, outil de persistance relationnel vers la plate-forme .NET. NHibernate (version 1.0) est compatible fonctionnellement avec Java Hibernate 2.1. C'est dans le monde .NET, un des outils de mapping les plus utilisés.

4. Les différentes approches du mapping

4.1 La persistance manuelle (ADO.NET, DataSet, …)

La persistance manuelle consiste à gérer soit même le mapping objet/relationnel. .NET propose dans ce domaine un large éventail d'outils. Le plus connu et le plus largement utilisé étant le DataSet.

Le DataSet est un objet qui réside en mémoire et qui correspond à une copie locale des données d'une base. Il contient les tables d'une base mais aussi les relations entre ces différentes tables et les contraintes appliquées aux données. XML étant utilisé sous .NET comme le standard de description et de persistance des données, l'objet DataSet est représenté sous cette forme. Les données d'un DataSet sont formalisées en XML et le schéma est écrit en XSD (XML Schema Definition Language).

L'objet DataSet étant déconnecté de toute source de données, il peut être utilisé pour la transmission de données entre différents tiers d'une application (rôle assuré auparavant par les Recordsets déconnectés sous ADO), mais aussi à d'autres systèmes grâce à sa représentation XML .
Si le DataSet convient parfaitement dans le cadre d'applications « simples », il présente de multiples inconvénients. Tout d'abord, un DataSet est « un objet de données ». A ce titre, il s'intègre relativement mal dans le cadre d'architectures multi-couches dans lesquelles les dépendances binaires sont importantes. En effet, un DataSet nécessitant l'utilisation de la DLL *System.Data*, toute manipulation côté client requiert l'intégration de cette DLL. Une approche qui va clairement à l'encontre des préceptes fondamentaux du multi-tiers.

De plus, un DataSet est une classe plutôt intrusive dans la mesure où le modèle du domaine est entièrement mélangé avec les méthodes techniques permettant de rechercher/sauvegarder les entités (factories).

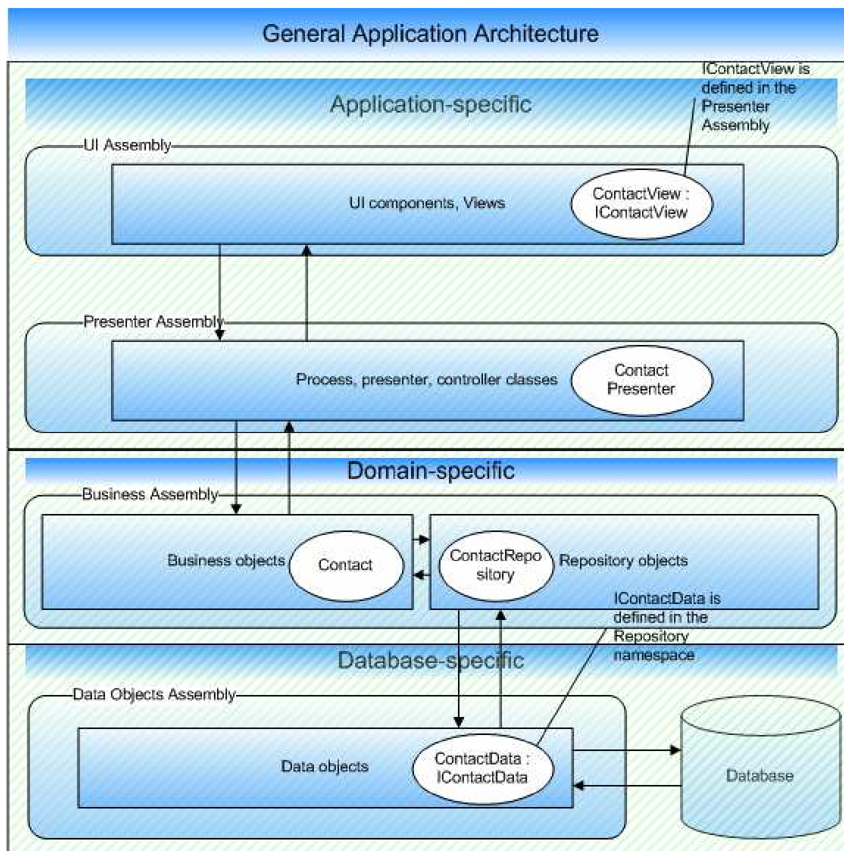
Tout cela fait qu'une approche ADO.NET manuelle doit être réservée à une application de faible envergure et dont le modèle du domaine (l'ensemble des classe persistantes) n'est pas stratégique (application essentiellement graphique, peu de métier, etc …)

4.2 La génération de code (CodeSmith, NetTiers, etc …)

L'approche par génération de code est une pratique censée rallier le meilleur des deux mondes (persistance manuelle et automatique). Cette approche consiste à générer **statiquement** tout le code permettant de rendre les services techniques habituels des outils de mapping (persistance automatique via recherche, sauvegarde, concurrence, etc …).

L'outil le plus en vogue dans la communauté .NET est CodeSmith : CodeSmith propose plusieurs modèles prédéfinis permettant de générer différents types d'applications. L'outil NetTier constitue la partie n-tiers de ces modèles. « *.NetTiers are CodeSmith templates for object-relational mapping*

that takes an existing SQLServer database and automatically generates a personnalized Data Tiers application block to use in your .Net applications."



L'inconvénient de l'approche par génération de code est de constituer en quelque sorte le parent pauvre du mapping. Certaines décisions ne peuvent être prises que dynamiquement (typiquement vérifier qu'un objet et ses propriétés sont « dirty »). L'approche par génération part du principe que l'utilisateur maîtrise lui-même le cycle de vie d'une entité.

4.2.1 La persistance automatique

La persistance automatique est à l'essence du mapping objet/relationnelle. Cette approche consiste à prendre en charge le cycle de vie d'une entité de manière dynamique (grâce à la génération de code et à la réfection). L'utilisateur ne se préoccupe à aucun moment de l'état de son entité.
La persistance automatique fournit les services suivants :

- recherche, sauvegarde, mise à jour
- concurrence
- cache de 1er niveau et second niveau
- navigation
- chargement à la demande
- transactions (éventuellement)

valtech days

5. Optimiser sa couche de persistance

Le fichier de mapping constitue la pierre angulaire d'un projet de mapping O/R. En fonction de son paramétrage, les performances pourront varier d'un facteur 100 voire quelque fois 1000 ! Dans la pratique, peu d'ouvrages ou documentations mettent en lumière les enjeux de ce fichier, ô combien important.

Dans tout processus de mapping, il y a certaines étapes à respecter. Parmi ces étapes, la mise en œuvre du fichier de configuration est une des plus importantes. En effet, une fois l'ensemble du modèle du domaine configuré, le paramétrage du mode "lazy loading" pour les collections et autres objets associés (par défaut nhibernate ou hibernate préchargent entièrement un graphe d'objets complet) va avoir un effet direct sur les performances. De cette manière, la navigation ou la récupération d'un graphe ne nécessitera pas le chargement de données inutiles.

Dans un exemple contenant une table Order possèdant une relation 0,n avec les lignes OrderLine. Si l'utilisateur est intéressé simplement par le nom du client facturé et la date de facturation, il n'est pas nécessaire de charger les données OrderLine de la facture. Qui plus est s'ils ont chacun des liens avec d'autres objets.
Dans un cadre purement objet, cette solution est assez naturelle à mettre en place car tout lien "tardif" ou "lazy loadé" est résolu lorsque l'utilisateur cherche à accéder à la dépendance. Exemple :

```
class LazyLoading {
  public void WorkWithOrderAndOrderLines(string orderID) {
       // Provoque le chargement d'un objet Order sans les OrderLine
       // SQL : Select * from Order where OrderID=?
       Order order = getSession().get(orderID, typeof(Order));

       // Provoque le chargement des OrderLine de l'order courant
       // SQL : Select * from OrderLine where OrderFK=?
       Collection orderLines = order.getOrderLines();
     }
}
```

Imaginons maintenant reproduire le même exemple pour N Orders avec une boucle. Le cas fonctionnel est trivial, nous souhaitons récupérer les factures avec un total supérieur à 1000 euros. A priori, il suffit d'implémenter une méthode CalculTotal() prenant en paramètre une collection de type OrderLine puis d'effectuer pour chaque ligne la multiplication Quantité*Prix, et ce pour chaque facture. Le tout est illustré dans le code source suivant.

```
class NPlusUnSelect {
   public void BestTotalOrder(string orderID) {
     // Provoque le chargement de l'ensemble des objet Orders de la table
     // Select * from Order 1 fois
     System.Collections.IList orders = getSession().find("from order");
        // Provoque le chargement des OrderLine de chaque order : N fois
        // Select * from OrderLine where OrderFK=?
        foreach (Orders ol in orders) {
           Collection orderLines = ol.getOrderLines();
           if (CalculTotal(orderLines) > 1000)
              Console.WriteLine("Good Order!!");
        }
   }
}
```

Rien ne présage dans un tel code du syndrome "N+1 Select" : le mal absolu en terme de mapping ! En pratique, pour une table contenant 1000 factures avec en moyenne 10 lignes Produit, il faudra compter pas moins de 1001 requêtes SQL pour exécuter cette simple tâche. Une requête pour récupérer l'ensemble des factures puis une autre pour chaque collection. **Un vrai désastre en terme de performance** surtout si on imagine multiplier ce facteur pour chaque utilisateur connecté. S'il peut paraître évident ici, le problème du N+1 Select est souvent masqué dans la complexité globale d'une application rendant délicate sa détection. Seule une analyse minutieuse des traces SQL (toujours activer le mode showSQL à true) permettra d'y remédier.

5.1 Quelles sont les solutions ?

La plupart des outils se sont attaqués très tôt à ce problème en proposant diverses optimisations. Hibernate fournit ainsi l'ordre Batch-Size permettant de précharger les n collections suivantes lorsque vous accédez à la première. Dans l'exemple précédent, un Batch-Size positionné à 10 aurait permit de diviser le facteur n+1 par 10 en regroupant dans la même requête SQL, 10 Collections appartenant à 10 entités Order différentes. Bien entendu, si votre application ne s'intéresse qu'à un Order, cette optimisation atteint vite son cas pire en ramenant plus d'enregistrement qu'il n'en faudrait.

Curieusement, la réponse idéale en terme de performance au problème du N+1 est une solution avec une philosophie "relationnelle". Calculer le total le plus élevé ou effectuer des calculs sur des tuples peut se résoudre en SQL ou HQL (le langage de requête objet d'hibernate) avec une seule requête :

```
HQL : select order from Order order join order.orderLines ol group by order having sum(ol.amount)
> 1000
```

Cette requête ne renvoie uniquement que les enregistrements pertinents sans précharger en mémoire un graphe d'objet entier. L'inconvénient est son écriture peu "objet" et son approche peu générique, pour ne pas dire contre-nature.

Malgré tout, entre les deux alternatives précédentes, il existe d'autres solutions plus élégantes même si légèrement moins performantes. La première concerne l'ordre outer-join qui, spécifié dans le fichier de mapping, permet d'indiquer à l'outil que l'association doit être résolue grâce à une jointure. Dans le cas d'une collection, hibernate va chercher à recréer systématiquement à partir d'une jointure l'association Order->OrderLines pour l'ensemble des entités Order de la table (via la requête suivante :

```
select order o, orderline ol where o.orderid=ol.orderfk).
```

Une approche extrêmement consommatrice en ressources mais qui a le mérite de réduire l'impact du Syndrome N+1.

L'autre solution, la plus optimale, consiste à tirer partie de l'API CreateCriteria pour précharger les collections en tirant partie d'un procédé appelé "Eager Fetching". Hibernate permet en effet de récupérer un graphe d'objet tout en spécifiant les associations à charger. Dans le cas des OrderLines, il suffit de spécifier explicitement qu'on souhaite récupérer toutes les collections de l'objet racine Order de la manière suivante :

```
CreateCriteria : List orders = session.createCriteria(Order.class).setFetchMonde("orderLines",
FetchMode.EAGER).list()
```

Hibernate réalise ainsi une jointure pour renvoyer la liste des Order avec une seule requête. Il reste cependant à extraire de manière distincte le jeu de résultats car les jointures ont tendance à renvoyer des tuples dupliqués.

```
class SelectOptimized {
        public void BestTotalOrder(string orderID) {
          // Provoque le chargement de l'ensemble des objet Orders de la table avec une requête JOIN
           HashMap orders =
                              New
HashMap(session.createCriteria(Order.class).setFetchMonde("orderLines",
FetchMode.EAGER).list());

            foreach (Orders ol in orders) {
                Collection orderLines = ol.getOrderLines();
                if (CalculTotal(orderLines) > 1000) Console.WriteLine("Good Order!!");
            }
        }
    }
```

Cet exemple illustre parfaitement le danger pour un utilisateur de naviguer dans un modèle objet au gré des besoins du moment pour récupérer un jeu de données. Il convient de toujours précharger les données avant de les manipuler.

Le problème du *N+1 Select* constitue 90% des problèmes de performances rencontrés sur le terrain.

6. Conclusion

Qu'elle soit manuelle ou automatique, la persistance est une des composantes clés d'une application. Un mauvais choix ou une mauvaise conception auront des conséquences irréversibles sur les performances. Plus que jamais il est nécessaire de comprendre que persistance transparente ne signifie pas persistance non maîtrisée.