

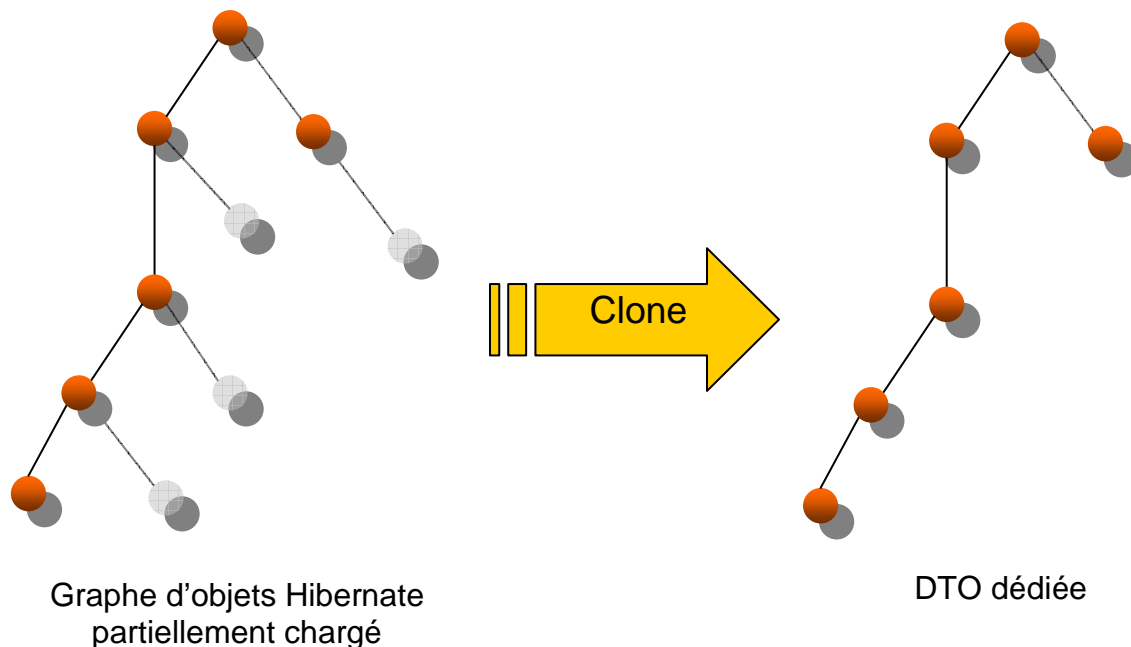
GWT : Hibernate inside...

Map of the problematique

Dans la vie, je suis d'un naturel obstiné (voire têtu les mauvais jours ;)). Le recours préconisé (jusque dans ce blog) aux DTO pour passer des objets du Domaine d'Hibernate à GWT ne me satisfait qu'à moitié pour les raisons suivantes :

- Le clonage des objets à un coût, même si j'ai tendance à ne pas le croire vraiment pénalisant
- Elle implique une double hiérarchie : celle des objets du Domaine et celle des DTO. Cette dernière est généralement une copie de la première, à laquelle est ajoutée une nuée de versions dégradées permettant d'éviter le problème des proxies non initialisés.

En effet, le mécanisme est généralement le suivant :



On voit bien ici qu'à chaque chargement partiel d'un graphe d'objet Hibernate doit correspondre une DTO dédiée !

Pire, le service GWT doit supporter plusieurs méthodes différentes pour une même action, en fonction des DTO qu'il peut recevoir.

Exemple :

```
public void updateUser(UserWithAddressDTO user) {...}
public void updateUser(CompleteUserDTO user) {...}
public void updateUser(SimpleUserDTO user) {...}
```

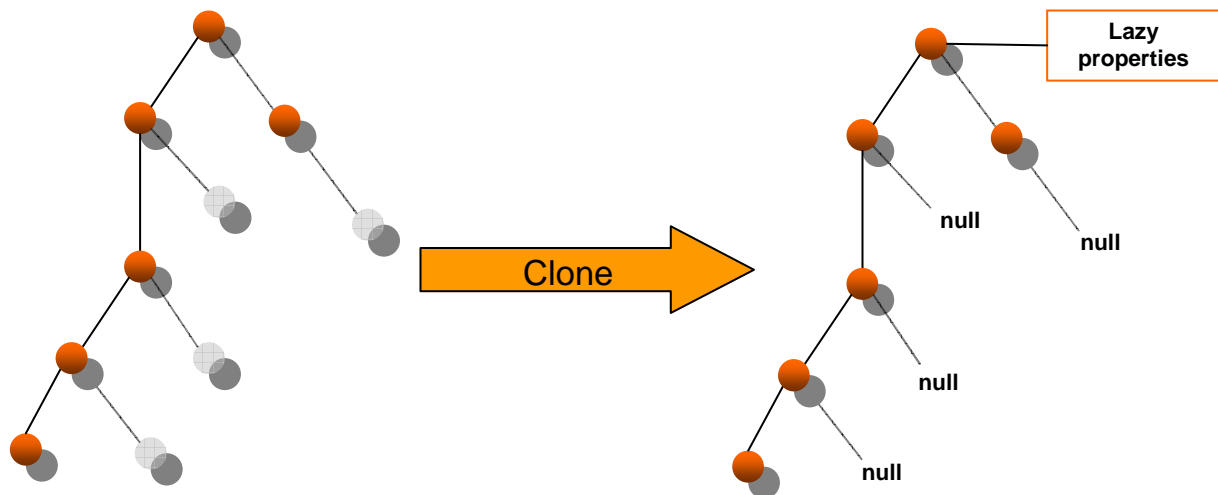
Cachez ce clone que je ne saurais voir...

L'idée directrice est d'envoyer les objets du Domaine (ce qui avec la version actuelle de GWT implique des limitations, nous y reviendrons) dans le code client de GWT afin d'éviter la profusion des DTO.

Pour cela, il nous d'abord faut régler le problème du lazy-loading. Comme me le faisait remarquer Emmanuel Bernard dans mon premier billet, le fait de passer les associations « lazy but not loaded » à *null* ne suffit pas : sans mécanisme supplémentaire, il devient alors impossible de distinguer une association nulle d'une autre non chargée. L'affaire est donc entendue : il nous faut marquer les associations non chargées. Pour faire simple, nous choisirons de stocker cette information sur le pojo lui-même.

Deuxième problème : `PersistentSet` et `java.sql.DateTime`. Ces classes ne sont pas reconnues par le compilateur JavaScript de GWT. Toujours pour rester simple et valider le concept, j'ai utilisé une librairie de clonage qui prendra en charge cette conversion. Dozer ? Non, pas possible... En effet, j'ai besoin d'une librairie à qui je puisse indiquer dynamiquement les propriétés à ne pas cloner (les associations non chargées et qui génèrent une `LazyInitializationException`), et Dozer ne prend pas en charge cette fonctionnalité. En fait, j'ai utilisé [beanLib](#) (une librairie qui gagnerait à être connue... si elle était documentée !), qui gère la duplication d'objets Hibernate et qui permet de déterminer à l'exécution les propriétés à cloner. J'en profite pour faire d'une pierre deux coups en remplissant ma liste des propriétés non chargée en même temps que je clone mon objet du Domaine.

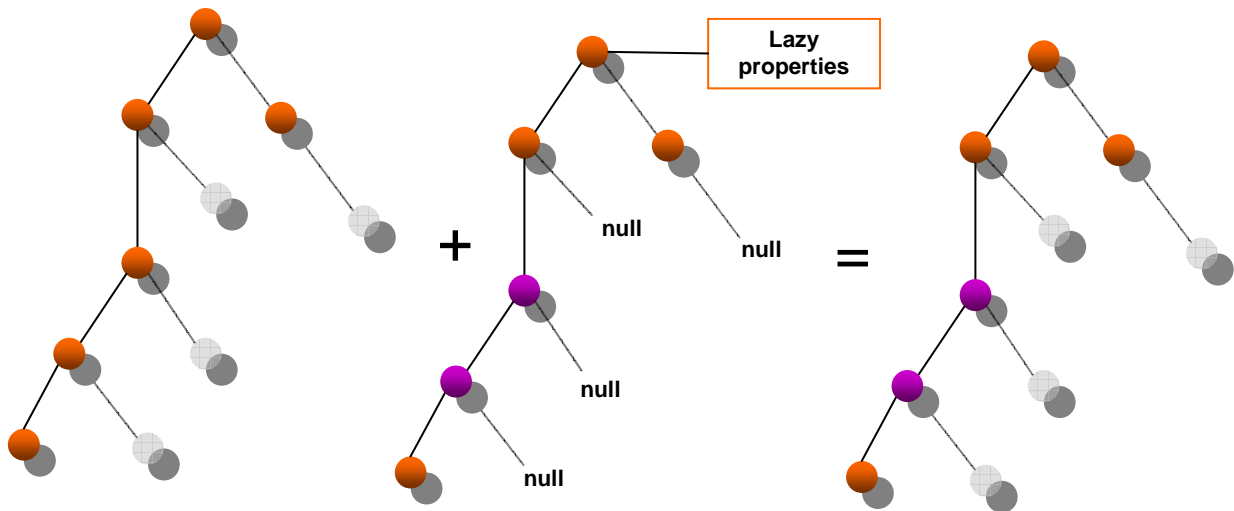
Le mécanisme peut donc se résumer ainsi :



Nous avons donc un POJO « neutralisé » que GWT sait parfaitement convertir en JavaScript.

Le chemin retour est un tout petit peu plus compliqué : en effet, pas question d'envoyer directement notre clone à la couche d'Accès au Données, sans quoi les propriétés non chargées seront remplacées par *null* par Hibernate puisqu'il n'y trouve plus ses proxies !

En fait, il convient de fusionner notre clone restitué par GWT avec un véritable objet Hibernate, en prenant soin de ne pas remplacer les proxies en place :



Il ne reste plus qu'à envoyer l'objet fusionné à la couche d'accès aux données qui mettra à jour la base de données.

Le prix à payer

Cette solution, relativement fonctionnelle, corrige le problème de duplication de la hiérarchie du Domaine et la profusion de DTO. Mais cela a un coût, lié notamment aux limitations de la version actuelle de GWT :

- Adhérence à l'interface `IsSerializable`
- Interface `Serializable` non supportée
- Les objets du Domaine doivent utiliser la syntaxe Java 1.4

Même s'il y a fort à parier que les prochaines versions de GWT dépasseront ces limitations, la version actuelle du LazyKiller doit s'y plier. En attendant donc qu'elles soient supportées par GWT, oubliez donc les collections typées et les annotations...

De plus, dans son état actuel, la librairie LazyKiller impose un héritage technique (LazyGwtPojo) afin de gérer les propriétés non chargées, et la couche service doit prendre en charge la conversion `Hibernate->clone` et `clone->Hibernate`.

Limitations

Dans mon esprit, le développement de la librairie LazyKiller est avant tout une preuve de concept me permettant de valider la faisabilité de l'envoi d'objets du Domaine à la couche GWT.

A ce titre, j'ai pris certains raccourcis de conception (clonage) et mes tests ont surtout couvert des cas basiques. J'ai notamment quelques interrogations sur la conversion d'une collection de proxys...

Dans le même ordre d'idée, je ne suis pas certain que la classe LazyKiller soit *thread-safe*.

Usage

Même si j'ai voulu rendre l'utilisation de la librairie la plus simple possible, l'utilisation en pratique nécessite un certains nombre d'étapes :

1. Ajouter lazy-killer.jar ainsi que ses dépendances (beanLib notamment) dans le classpath du projet. Il faut aussi modifier les fichiers de commande générés par GWT pour l'y ajouter, sans quoi les packages 'lazykiller' ne seront pas trouvés.
2. Ajoutez `<inherits name='org.dotnetguru.lazykiller.LazyKiller' />` dans le fichier `<MonModule>.gwt.xml`
3. Vos classes du domaine doivent hériter de LazyGwtPojo (il s'agit d'une classe abstraite gérant les « lazy properties » et implémentant l'affreux `IsSerializable`)

```
public class User extends LazyGwtPojo
```

4. Modifiez votre couche de service comme suit :

Mode stateful

Envoi d'un objet du domaine

```
// Get Hibernate POJO
//
User user = <your_service>.getUser(id);

// Lazy killing
//
User cloneUser = (User) new LazyKiller().detach(user);

// Store Hibernate POJO in HTTP session
//
getThreadLocalRequest().getSession().setAttribute("User", user);

// Return cloned POJO
//
return cloneUser;
```

Réception d'un objet du domaine

```
// Get Hibernate user from HTTP Session
//
User hibernateUser = (User)
    getThreadLocalRequest().getSession().getAttribute("User");

// Merge the GWT pojo with the Hibernate one
//
new LazyKiller().attach(hibernateUser, user);

// Update the user in base
//
<your_service>.updateUser(hibernateUser);
```

Mode stateless

Envoi d'un objet du domaine

```
// Get Hibernate POJO
//
User user = <your_service>.getUser(id);

// Lazy killing
//
return (User) new LazyKiller().detach(user);
```

Réception d'un objet du domaine

```
// Get a fresh Hibernate POJO
```

```
//  
User hibernateUser = <your_service>.getUser(id);  
  
// Merge the GWT pojo with the Hibernate one  
//  
new LazyKiller().attach(hibernateUser, user);  
  
// Update the user in base  
//  
<your_service>.updateUser(hibernateUser);
```

A venir

Il existe plusieurs pistes pour améliorer ce petit bout de code (si tant est qu'il soit d'un grand intérêt ;-)) :

- Rendre le LazyKiller *thread-safe*,
- Déplacer le clonage avant et après l'appel aux services GWT, en s'appuyant / s'inspirant de ce qui a été fait pour le GWTHandler,
- Supprimer le clonage ? Est-il possible de remplir la table des propriétés non chargées directement à la compilation Java->JavaScript ? Cela sous-tend en outre de reprendre les conversions effectuées par beanLib...
- Supprimer l'héritage technique ? Il s'agit pour l'instant d'un vœu pieux, je n'ai encore aucune idée technique de la faisabilité de la chose...

Conclusion

LazyKiller est une preuve de concept : oui, il est possible d'envoyer des objets Hibernate à la couche de présentation GWT sans passer par des DTO explicites. Cependant, il existe à ce jour pas mal de restrictions qui contraignent son usage (l'héritage technique et la syntaxe Java 1.4 en sont les plus gênants).

En espérant que les prochaines versions de GWT corrigeront ces limitations !

Bruno Marchesson