CHAPTER 1

Persistance des données de TinyBlog avec Voyage et Mongo

1.1 Correction semaine précédente

Vous pouvez charger la correction de la semaine précédente en exécutant le code suivant:

```
Metacello new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    version: #week3solution;
    configuration: 'TinyBlog';
    load
```

Puis, testez le code en exécutant:

```
TBTeapotWebApp start
```

Attention, vérifiez d'abord que votre propre serveur HTTP ne soit pas déjà lancé sur le même port . Si c'est le cas, arrêtez votre application ou changez son port.

Vous devez ajouter des posts dans le blog pour voir quelquechose.

```
TBBlog reset ; createDemoPosts
```

Avant de passer à la suite, stoppez votre serveur Teapot:

```
TBTeapotWebApp stop
```

1.2 Préparation de sauvegarde extérieure avec Voyage

Avoir un modèle d'objets en mémoire fonctionne bien, et sauvegarder l'image Pharo sauve aussi ces objets. Toutefois, il est préférable de pouvoir sauver les objets dans une base de données extérieure. Voyage permet de sauver les objets dans une base de donnée Mongo. Voyage est un frameworks qui propose la même API a plusieurs bases de données documents comme Mongo ou UnQLite.

C'est ce que nous allons voir. Dans un premier temps, nous allons utiliser la capacité de Voyage à simuler une base extérieure. Ceci est très pratique en phase de développement. Dans un second temps, nous installerons une base de données Mongo et nous y accéderons à travers Voyage.

1.3 Configurer Voyage pour sauvegarder des objets TBBlog

Grâce à la méthode de classe `isVoyageRoot`, nous déclarons que les objets de la classe `TBBlog` doivent être sauvés dans la base en tant qu'objets racines.

```
TBBlog class >> isVoyageRoot
   "Indicates that instances of this class are top level document in
    noSQL databases"
   ^ true
```

Nous devons ensuite soit créer une connexion sur une base de données réelle soit travailler en mémoire. C'est cette dernière option que nous choisissons pour l'instant en utilisant cette expression.

```
VOMemoryRepository new enableSingleton.
```

Le message `enableSingleton` indique à Voyage que nous n'utilisons qu'une seule base de donnée ce qui nous permet de ne pas avoir à préciser avec laquelle nous travaillons.

Nous définissons une méthode `initializeVoyageOnMemoryDB` dont le rôle est d'initialiser correctement la base.

```
TBBlog class >> initializeVoyageOnMemoryDB
   VOMemoryRepository new enableSingleton
```

Ici nous n'avons pas besoin de stocker la base dans une variable d'instance car nous n'avons qu'une seule base de donnée en mode singleton.

Nous redéfinissons les méthodes de classe `reset` et `initialize` pour nous assurer que l'endroit où la base est sauvée est réinitialisé lorsque l'on charge le code.

La méthode `reset` réinitialize la base.

```
TBBlog class >> reset
      self initializeVoyageOnMemoryDB

TBBlog class >> initialize
      self reset
```

Le cas de de la méthode `current` est plus délicat. Avant l'utilisation de Mongo, nous avions un singleton tout simple. Cependant utiliser un Singleton ne fonctionne plus car imaginons que nous ayons sauvé notre blog et que le serveur s'éteigne par accident ou que nous rechargions une nouvelle version du code. Ceci conduirait à une réinitialisation et création d'une nouvelle instance. Nous pouvons donc nous retrouver avec une instance différente de celle sauvée.

Nous redéfinissons `current` de manière à faire une requête dans la base. Comme pour le moment nous ne gérons qu'un blog il nous suffit de faire `self selectOne: [ :each | true ]` ou `self selectAll anyOne`. Nous nous assurons de créer une nouvelle instance et la sauvegarder si aucune instance n'existe dans la base.

```
TBBlog class >> current
   ^ self selectAll
      ifNotEmpty: [ :x | x anyOne ]
      ifEmpty: [ self new save ]
```

La variable `uniqueInstance` qui servait auparavant à stocker le singleton `TBBlog` peut être enlevée.

1.4 Sauvegarde d'un blog

Nous devons maintenant modifier la méthode `writeBlogPost:` pour sauver le blog lors de l'ajout d'un post.

```
TBBlog >> writeBlogPost: aPost
   "Write the blog post in database"
  self allBlogPosts add: aPost.
   self save
```

Nous pouvons aussi modifier la méthode `remove` afin de sauver le nouvel état d'un blog.

```
TBBlog >> removeAllPosts
   posts := OrderedCollection new.
   self save.
```

1.5 Utilisation de la base

Alors même que la base est en mémoire et bien que nous pouvons accéder au blog en utilisant le singleton de la classe `TBBlog`, nous allons montrer l'API

offerte par Voyage. C'est la même API que nous pourrons utiliser pour accéder à une base mongo.

Nous créons des posts ainsi :

```
TBBlog createDemoPosts.
```

Nous pouvons compter le nombre de blog sauvés. `count` fait partie de l'API directe de Voyage. Ici nous obtenons 1 ce qui est normal puisque le blog est implémenté comme un singleton.

```
TBBlog count
>1
```

De la même manière, nous pouvons sélectionner tous les objets sauvés.

```
TBBlog selectAll
```

On peut supprimer un objet racine.

```
TBBlog current remove
```

Vous pouvez voir l'API de Voyage en parcourant

- la classe `Class`, et
- la classe `VORepository` qui est la racine d'héritage des bases de données en mémoire ou extérieure.

Ces queries sont plus pertinentes quand on a plus d'objets mais nous ferions exactement les mêmes.

1.6 Si nous devions sauvegarder les posts

Cette section n'est pas à implémenter et elle est juste donnée à titre d'exemple plus d'explications sont données dans le chapitre sur Voyage dans Enterprise Pharo disponible a http://www.pharo.org. Nous voulons illustrer que déclarer une classe comme une racine Voyage a une influence sur comment une instance de cette classe est sauvée et rechargée. En particulier car elle ne fait plus partie de l'objet qui lui faisait référence : dans la base elle n'est plus une souspartie du document représentant l'objet qui la référence mais un document autonome.

Nous pourrions définir qu'un post soit un élément qui peut être sauvegardé de manière autonome. Cela permettrait de sauver des posts de manière indépendante d'un blog. Par contre, si nous représentions les commentaires d'un post nous ne les déclarerions pas non plus comme racine car sauver ou manipuler un commentaire en dehors du contexte de son post ne fait pas beaucoup de sens.

Lorsqu'un post n'est pas une racine, vous n'avez pas la certitude d'unicité de celui-ci lors du chargement depuis la base. En effet, lors du chargement (et

ce qui peut être contraire à la situation du graphe d'objet avant la sauvegarde) un post n'est alors pas partagé entre deux instances de blogs. Si avant la sauvegarde en base un post était partagé entre deux blogs, après le chargement depuis la base, ce post sera dupliqué car recréé à partir de la définition du blog (et le blog contient alors complètement le post).

Post comme racine = Unicité

Si vous désirez qu'un post soit partagé et unique entre plusieurs instances de blog, alors les objets `TBPost` doivent être déclarés comme une racine dans la base. Lorsque c'est le cas, les posts sont sauvés comme des entités autonomes et les instances de `TBBlog` feront référence à ces entités au lieu que leurs définitions soient incluses dans celle des blogs. Cela a pour effet qu'un post donné devient unique et partageable via une référence depuis le blog.

Pour cela on nous définirions les méthodes suivantes:

```
TBPost class >> isVoyageRoot
   "Indicates that instances of this class are top level document in
    noSQL databases"
   ^ true
```

Lors de l'ajout d'un post dans un blog, il est maintenant important de sauver le blog et le nouveau post.

```
TBBlog >> writeBlogPost: aPost
   "Write the blog post in database"
   posts add: aPost.
   aPost save.
   self save
```

```
TBBlog >> removeAllPosts
   posts do: [ :each | each remove ].
   posts := OrderedCollection new.
   self save.
```

Ici dans la méthode `removeAllPosts` nous enlevons chaque posts puis nous remettons à jour la collection.

1.7 Tester avec une base Mongo

Nous allons maintenant utiliser une base Mongo externe à Pharo. En utilisant Voyage nous pouvons rapidement sauver nos posts dans une base de données Mongo. Cette section explique rapidement la mise en oeuvre et les quelques modifications que nous devons apporter à notre projet Pharo pour y parvenir.

1.8 Installation de Mongo

Quel que soit votre système d'exploitation (Linux, Mac OSX ou Windows), vous pouvez installer un serveur Mongo localement sur votre machine. Cela est pratique pour tester votre application sans avoir besoin d'une connexion Internet. Nous ne détaillons pas ici comment installer Mongo sur votre système, référez-vous à la documentation Mongo.

> **Note** Le serveur Mongo ne doit pas utiliser d'authentification car la nouvelle méthode de chiffrement SCRAM utilisée par MongoDB 3.0 n'est actuellement pas supportée par Voyage.

Cependant vous devez créer une base de données nommée 'tinyblog'. Lorsque, vous avez installé et lancé un serveur Mongo localement, vous pouvez y accéder depuis Pharo.

1.9 Connexion à un serveur local

Nous définissions les méthodes `initializeLocalhostMongoDB` pour établir la connexion vers la base de données.

```
TBBlog class >> initializeLocalhostMongoDB
   | repository |
   repository := VOMongoRepository database: 'tinyblog'.
   repository enableSingleton.
```

Il faut aussi s'assurer de la ré-initialisation de la connexion à la base lors du reset de la classe.

```
TBBlog class >> reset
   self initializeLocalhostMongoDB
```

En cas de problème

Notez que si vous avez besoin de réinitialiser la base extérieure complètement, vous pouvez utiliser la méthode `dropDatabase`.

```
(VOMongoRepository
   host: 'localhost'
   database: 'tinyblog') dropDatabase
```

Attention : Changements de TBBlog

Si vous utilisez une base locale plutôt qu'une base en mémoire, à chaque fois que vous ajoutez une nouvelle racine d'objets ou modifiez la définition d'une classe racine (ajout, retrait, modification d'attribut) il est capital de réinitializer le cache maintenu par Voyage. La réinitialisation se fait comme suit:

```
VORepository current reset
```

1.10 Conclusion

Voyage propose une API sympathique pour gérer de manière transparente la sauvegarde d'objets soit en mémoire soit dans une base de données document. Votre application peut maintenant être sauvée dans la base et vous êtes donc prêt pour construire son interface web.