TP 2 : Etat de l'existant et poursuite du projet

M2 MBDS

19/11/2013

Présentation Générale

L'objectif de ce premier TP est de stabiliser votre base de projet et d'ajouter de nouvelles fonctionnalités.

Nous allons notamment commence par corriger le problème qui a du vous bloquer dans le TP d'hier.

Objectif

1/ Etat de l'existant

Vous devriez donc avoir 3 classes :

User :

```
class User {
    // Attributs de classe

    // Une relation one to many vers les classes Message et Groupe
    // User peut avoir plusieurs messages
    // User peut avoir plusieurs groupes
    static hasMany = [messages:Message,groupes:Groupe]

    static constraints = {
        // Contraintes
    }
}
```

Groupe :

```
class Groupe {
    // Attributs de classe

    // Une relation one to many car un groupe peut contenir
    // plusieurs utilisateurs
    static hasMany      = [users:User]

    static constraints = {
        // Contraintes
    }
}
```

Message :

```
class Message {
    // Attributs de classe

    // On défini ici que Message est le côté "faible"
    // de la relation User - Message
    static belongsTo = [User]

    static constraints = {
        // Contraintes
    }
}
```

Nous allons maintenant essayer de comprendre ce qui ne va pas.

Pour cela, nous allons faire en sorte d'utiliser une base de donnée type « MySQL » pour avoir un PhpMyAdmin nous permettant d'avoir une vue claire de notre base de donnée et du modèle créé.

2/ Utilisation d'une base de donnée type « MySQL »

Je vais vous donner ici les éléments pour accomplir cette tâche.

Si vous préférez utiliser un autre SGBD, libre à vous !

1) Installer MySQL
2) Télécharger le plugin Grails qui contient le connecteur MySQL
   - Allez sur cette page : http://grails.org/plugin/mysql-connectorj
   - Récupérez la commande en tête de page pour ajouter cette dépendance à votre BuildConfig.groovy, validez la configuration dans le tooltip qui apparait
   - Exécutez un « compile » de votre projet afin de télécharger la nouvelle dépendance
3) Editez votre fichier « DataSource.groovy » pour le faire pointer sur votre nouvelle base de données :

```
development {
    dataSource {
        dbCreate = "create-drop"
        url = "jdbc:mysql://localhost/cwitter"
        driverClassName = "com.mysql.jdbc.Driver"
        username = "root"
        password = ""
    }
}
```

4) Créez votre base de données, Grails crée les tables mais pas la base, en l'occurrence ici la base est nommée « cwitter »
5) Relancez votre projet. Si vous voyez vos tables se créer, tout est bon !

3/ Constatations

Observez votre base de données, une chose doit vous sauter aux yeux en pensant au modèle dont vous auriez besoin pour réaliser ce que vous devez faire et le modèle que vous avez actuellement.

Point de validation : Appelez-moi pour me faire part de vos remarques à ce sujet.

Nous allons maintenant créer quelques objets pour confirmer nos doutes.

Allez dans ***BootStrap.groovy*** :

Créons un Objet de chaque type, voici les syntaxes à utiliser :

```
def userInstance =
        new User(/*mettez ici les propriétés necessaires pour la création d'un utilisateur*/)


def groupeInstance =
        new Groupe(/*mettez ici les propriétés necessaires pour la création d'un utilisateur*/)


def messageInstance =
        new Message(/*mettez ici les propriétés necessaires pour la création d'un utilisateur*/)

```

Si vous recopiez simplement ces lignes, vos objets ne se créeront pas et ce pour deux raisons :

- Il existe des dépendances entre vos objets qui doivent être crées dans un ordre précis
- Il faut donner des ordres de persistance pour qu'Hibernate sauvegarde vos objets

Reportez-vous à votre cours pour trouver comment créer vos objets.

Une fois vos objets crées vous devriez confirmer votre hypothèse relative au problème du modèle de données.

Point de validation : Appelez-moi pour me faire part de vos remarques à ce sujet.

4/ Corrections

Nous allons maintenant corriger notre modèle afin de modéliser ce que nous voulons.

Dans un premier temps nous allons devoir nommer la relation d'appartenance présente dans la classe Groupe.

Faire comme ceci :

```
12      static belongsTo    = [owner:User]
```

De cette manière, nous avons donc un attribut qui représente la relation inverse.

Jusque-là nous avions uniquement :

Groupe.class :

```
6       static hasMany      = [users:User]
```

User.class :

```
30      static hasMany = [messages:Message groupes:Groupe]
```

Cette syntaxe représente pour Grails une simple relation « many to many », c'est pour cette raison que vous n'aviez pas le modèle de données que vous attendiez.

Nommer la relations inverse (owner :User) va nous permettre de spécifier dans la classe User qu'il ne s'agit pas d'une relation « many to many » mais en fait de deux relations « one to many » distinctes.

Pour se faire utiliser l'instruction suivante dans User.class :

```
static mappedBy = [groupes: "owner"]
```

Ceci va spécifier à Grails et donc à GORM que l'attribut « groupes » déclaré dans le « hasMany » de la classe User dois être mappé de sur l'attribut « owner » de la classe Groupe.

Relancez votre projet afin de recréer les tables comme il faut et constatez le changement.

Point de validation : Appelez-moi pour me montrer le modèle modifié

5/ Ajout de données

Modifiez le « BootStrap.groovy » afin d'ajouter :

- 5 utilisateurs différents
- 1 – 3 groupes pour chaque utilisateur contenant au moins un utilisateur
- 2 – 5 messages pour chaque utilisateur

Point de validation : Appelez-moi pour me montrer les données générées

---

6/ Construction de pages

1) User Home

Construisez une page en vous inspirant des contrôleurs et vues générées respectant les contraintes suivantes :

- url d'accès pour la page : **/cwitter/user_home/id_user**
- Différents bloc qui présenteront
  - Les informations personnelles de l'utilisateur
  - Les messages de l'utilisateur courant
  - Les messages des utilisateurs dans les groupes possédés par l'utilisateur courant
  - Une liste des utilisateurs dans les groupes possédés par l'utilisateur courant
  - Une liste de tous les messages (ceux de l'utilisateur et ceux des utilisateurs dans ses groupes) triés par ordre inverse de création

---

Point de validation : Appelez-moi pour me montrer la page créée

---

2) Gestion des groupes

Construisez une page permettant de créer des groupes et d'y ajouter des utilisateurs :

- url d'accès pour la page : **/cwitter/user_groups/id_user**
- Différents bloc qui présenteront
  - Les différents groupes existants et les utilisateurs contenus
  - Un « formulaire » permettant de créer un groupe (saisir toutes les données nécessaire) et de sélectionner des utilisateurs via la méthode de votre choix parmi :
    - Un champ de recherche / une page supplémentaire
    - Un champ de recherche avec une propriété autocomplete (Ajax) (Bonus)
    - Autre méthode de votre choix plus ergonomique (Bonus)
  - Un « formulaire » permettant de :
    - Modifier un groupe existant
      - Informations du groupe
      - Utilisateurs contenus dans les groupes via la méthode de votre choix parmi :
        - Des checkbox / boutons
        - Un système de Drag'n Drop (Bonus)

Point de validation : Appelez-moi pour me montrer la/les page(s) créée

Rendu

Ce projet sera noté à la fin des 3 séances et constituera une part importante de votre notation dans ce module.

Si vous avez des questions, adressez-moi un mail à l'adresse greg.galli@tokidev.fr, je ferai de mon mieux pour revenir vers vous le plus rapidement possible.

Des bonus seront accordés aux groupes qui feront le choix d'implémenter les parties « Bonus »

Un design soigné et une bonne ergonomie sont des petits plus qui seront pris en compte dans la notation de votre projet sans l'impacter de manière dramatique.