RAILS

EXERCICE PRATIQUE SUR LES ASSOCIATIONS

PARTIE 1 – ASSOCIATIONS 1-1

ELS – MARS 2014

1 OBJECTIFS

Mettre en œuvre une petite application Rails s'appuyant sur les 3 types d'associations : 1-1, 1-plusieurs et plusieurs-plusieurs.

Le travail consiste essentiellement à réaliser un « échafaudage ».

Les vues seront traitées de la manière la plus sommaire qui soit, sans feuilles de styles, sans Ajax, en s'appuyant sur les vues générées par le « scaffold statique», puis modifiées et complétées de manière à offrir les fonctionnalités désirées.

De manière générale, les contrôleurs, modèles et vues seront réalisés de sorte à :

- pouvoir afficher le contenu des tables (list & show)
- pouvoir en modifier le contenu (edit)
- pouvoir créer de nouvelles lignes (new)

Tout en garantissant une cohérence des informations contenues dans la base de données, notamment pour ce qui concerne la valeur des clés étrangères (foreign keys).

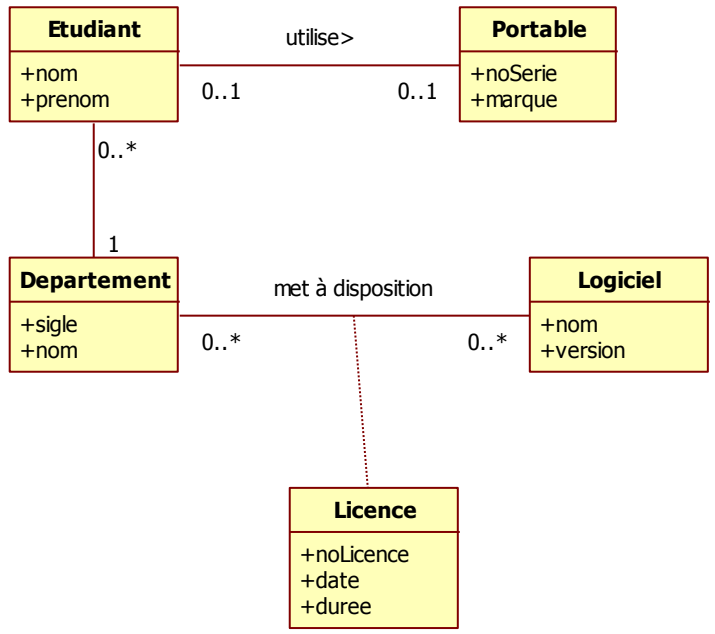
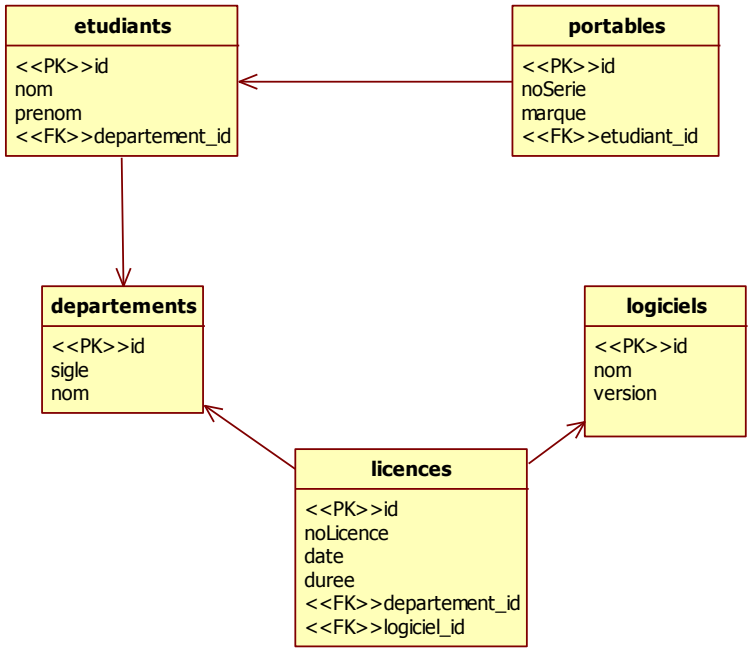
2 APPLICATIF A REALISER

Gestion des portables & licences au sein d'une école

Description

On supposera que le service informatique de l'école met à disposition de chaque étudiant un ordinateur portable sur lequel sont installés de base un certain nombre de logiciels.

La liste de ces logiciels dépend directement du département dans lequel se trouve inscrit l'étudiant. Seront installés sur son portable tous les logiciels dont le département possède une licence. Les licences ne sont accordées que pour un seul département (simplification).

Modèle conceptuel	
Modèle relationnel	

3 CREATION DU PROJET

CREATION DE LA BASE DE DONNEES

Au départ, créer simplement la base de données (p.e. : gesport) sans aucune table. Ces dernières seront créées en utilisant le concept des « migrations », voir plus loin.

CREER LE PROJET RAILS (P.E. : GESPORT)

```
c:\>rails new gesport -d mysql
```

⇨ Création du projet dans le dossier c:\gesport

CONFIGURER RAILS

Compléter le fichier `C:\gesport\config\database.yml` en spécifiant le nom de la base de données et le mot de passe.

4 CREER LA RESSOURCE ETUDIANT, SON MODELE, CONTROLEUR, VUES ET TABLE

Les étudiants seront limités au départ aux deux seuls champs `nom` et `prenom`. Sa clé étrangère sera introduite dans le cadre de la mise en place de l'association n-n.

4.1 DEMARCHE GENERALE

La démarche décrite ci-dessous pourra être appliquée pour chacun des types de ressources utilisés par l'application.

La démarche s'appuie sur **le concept des « migrations »** (Voir annexe).

Etape 1/ Génération de l'échafaudage (contrôleur, vues), du modèle et du fichier de migration

```
c:\gesport>rails g scaffold Etudiant prenom:string nom:string
```

⇨ Création, entre autres :

- du fichier de migration `xxx_create_etudiants.rb`
- du modèle `Etudiant.rb`
- du contrôleur `etudiants_controller.rb`
- des 4 vues `edit, show, new et index`

Etape 2/ Lancer la migration

```
c:\gesport>rake db:migrate
```

⇨ Création de la table dans la base de données

Etape 3/ Charger des données de test dans la table `etudiants`

Créer 3 ou 4 étudiants..

Comment ? En utilisant un mécanisme ad-hoc, très pratique pour mettre en place des données de test.

Se référer à l'Annexe « Migrations » et consultez le paragraphe [7.2 Chargement de données de garniture|

Etape 4/ Tester au moyen du serveur WEBrick et un navigateur

```
>http://127.0.0.1:3000/etudiants
```

4.2 OBSERVATION ET CONTROLES

Une fois la ressource créée..

Consulter le fichier `C:\gesport\db\schema.rb`, qui contient l'état actuel de la base de données (ce fichier est mis à jour à chaque exécution d'une migration).

☺ *Ce fichier pourrait être utilisé pour recréer toute la base de données depuis scrash.*

```
ActiveRecord::Schema.define(:version => 20140318094347) do
```

➔ *No de version actuel, le même que la valeur actuelle indiquée dans la table* `schema_migration` *de la base de données*

```
  create_table "etudiants", :force => true do |t|
    t.string   "prenom"
    t.string   "nom"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

end
```

Enfin, consulter les fichiers générés par le scaffold (contrôleur et vues associées)

5 LA RESSOURCE PORTABLE

Utiliser la même démarche que pour la ressource Etudiant.

- La table `portables` comportera deux champs : `no_serie` et `marque`.
- La table `portables` comporte en plus une clé étrangère : **etudiant_id**

Pour respecter les conventions de rail, les clés étrangères doivent obéir à la forme « **nomModele_id** ». Il est toutefois possible de contourner cette convention (Voir le Chapitre [ORM ActiveRecord])..

⇨ Voici la commande scaffold qui permettra au passage de spécifier la clé étrangère :

```
>rails g scaffold Portable no_serie:string marque:string etudiant_id:integer
```

Voici la migration générée :

```
class CreatePortables < ActiveRecord::Migration
  def change
    create_table :portables do |t|
      t.string :no_serie
```

```
      t.string :marque
      t.integer :etudiant_id

      t.timestamps
    end
  end
end
```


Introduisez des données de test dans la table portable. Opérez toujours en utilisant le mécanisme des garnitures.


Ouvrez une console de test sur la base de données. Placez-vous dans le répertoire de l'applicatif et saisissez :

```
> rails c
```

La console de test vous place dans l'environnement des modèles de l'application et vous permet de saisir directement des commandes Ruby.

Saisissez les commandes suivantes..

```
Portable.all
```

➔ Retourne le tableau de tous les portables enregistrés

```
Portable.create(:id => 4, :no_serie=>"ttt-dd123", :marque=>"Mac")
```

➔ Créer nouveau portable, immédiatement sauvegardé dans la base de données

```
Portable.where("no_serie='ttt-dd123'")
```

➔ Retourne le dernier portable enregistré

```
Portable.count
```

➔ Retourne le nombre de portables enregistrés

```
Etudiant.first
```

➔ Retourne premier des étudiants enregistrés (avec `id==1`)

 Puis, toujours dans la console de test, essayez d'obtenir le portable du premier étudiant en écrivant :

```
Etudiant.first.portable
```

➔ `Etudiant.first` retourne le premier étudiant
à qui on envoie le message `portable`, dans l'objectif de retourner son portable ..

Réponse ?..

 `NoMethodError: undefined method `portable'`

La méthode portable n'existe pas..

⇨ L'association 1..1 entre Etudiant et Portable n'a pas encore été mise en place..

⇨ Passer à l'étape suivante..

6 ASSOCIATION 1-1 : ETUDIANT <> PORTABLE

Vous allez compléter le code des modèles **`Etudiant.rb`** et **`Portable.rb`** de manière à mettre en œuvre de manière correcte la relation conceptuelle 1-1 existant entre le modèle `Etudiant` et le modèle `Portable`.

 Référer à la théorie du chapitre [ORM - ActiveRecord]).

 Puis essayez à nouveau avec la console de test..

 Attention !! Quand on modifie les classes Model, il est nécessaire de relancer la console de test (`EXIT` et `rails c`)

Testez :

```
Etudiant.first.portable

Portable.all[0].etudiant.prenom

Portable.all[1].etudiant.prenom
```

Etc..

6.1 AJOUTER UN CONTRAINTE D'INTEGRITE SUR LA CLE ETRANGERE

La technologie des migrations ne prévoit pas d'instruction spéciale à même de spécifier une contrainte d'intégrité accompagnant une clé étrangère. En effet, le contrôle d'intégrité peut très bien être opéré par l'application elle-même (contrôler que la valeur de la clé étrangère correspond à un identificateur valide de la table référencée), et nombre de programmeurs omettent de le préciser au niveau même de la base de données. Il peut être recommandé de le faire dans certains cas.

Un fichier de migration permet d'envoyer directement du SQL au moyen d'une instruction « `execute` » que l'on peut rajouter dans la méthode « `change` »

```
def change
        create_table "portables" do |t|
            etc..
         end
         execute "alter table portables
               add constraint fk_portables_etudiants
               foreign key (etudiant_id)  references etudiants(id)"

end
```

Testez..

Etape1 : créer une migration spécifique

```
>rails g migration ajout_contrainte_portables
```

➔Création d'un fichier de migration `xxxx_ajout_contrainte_portables.rb`

Etape2 : Editez ce fichier (en écrivant la méthode SQL et exécutez la migration (`rake db :migrate`)

Note : pour la méthode `self.down`, supprimez la contrainte avec :

```
execute "alter table portables
      drop foreign key fk_portables_etudiants"
```

 Attention !! L'ajout de cette contrainte ne sera acceptée par MySQL que si toutes les lignes de la table portables ont une clé étrangère etudiant_id référencent bien un étudiant existant !!

Etape3 : Contrôlez que la contrainte a bien été enregistrée

a) *au moyen de votre client MySQL*
b) *en testant la contrainte au moyen de la console de test, avec un `id` d'étudiant qui n'existe pas, comme par exemple :*

```
Portable.create(:etudiant_id=>10)
```

6.2 CONTROLES DE VALIDATION DE SAISIE POUR ETUDIANTS ET PORTABLES

Se référer à l'annexe du document (Validation de saisie), et faire en sorte:

☞ Que le `nom` de l'étudiant soit obligatoirement saisi lors de la création d'un nouvel étudiant

☞ Que le no de série d'un portable soit obligatoirement saisi lors de la création d'un nouveau portable

☞ Que le no de série d'un portable soit unique

 *Testez dans la console de test, saisissez*

```
p=Portable.new(:etudiant_id=>1)
p.save
```

→ `false` (la sauvegarde n'a pas pu être opérée en raison de l'absence du no de série)

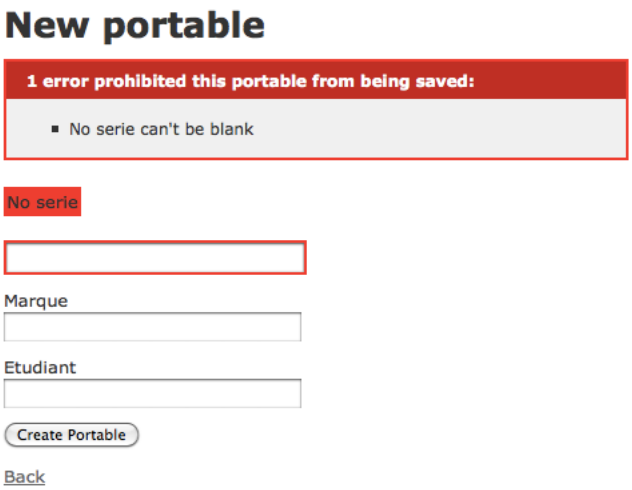
```
p=Portable.new(:no_serie=>11, :etudiant_id=>1)
p.save
```

→ `true` (la sauvegarde a bien eu lieu)

```
p=Portable.new(:no_serie=>11, :etudiant_id=>1)
p.save
```

→ `false` (la sauvegarde n'a pas pu être opérée en raison de la non unicité du no de série)

Testez l'applicatif, créez un portable sans indiquer de no de série, vous obtiendrez :



Le formulaire d'édition ou de création généré par le scaffold utilise la méthode « `form_for..` ». Cette dernière opère un contrôle des champs au moyen d'une requête Ajax opérée en arrière plan dès que l'utilisateur clique sur le bouton de soumission du formulaire et affiche les messages d'erreur.

☺ ***Solution : Voir étape 1***

6.3 AMELIORER L'INTERFACE UTILISATEUR - COMPLETER LES SCAFFOLD

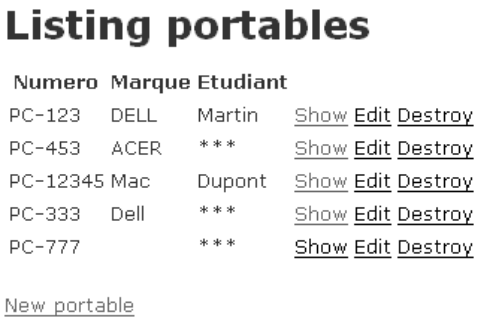
⇨ AMELIORATION NO1 - DU COTE PORTABLES

Action `show` ⇨Afficher le nom de l'étudiant associé – Si le portable n'est pas attribué, afficher « Non attribué »

☺ ***Solution : Voir étape 2***

⇨ AMELIORATION NO2 - DU COTE PORTABLES

Action `index` : Idem, afficher le nom de l'étudiant associé pour chaque portable et « *** » pour les portables non attribués.



☺ ***Solution : Voir étape 3***

⇨ AMELIORATION NO3 - DU COTE PORTABLES

On va maintenant travailler sur l'édition d'un portable en rajoutant un « combo-box » qui permette de sélectionner un étudiant parmi tous les étudiants inscrits.

Remplacer le champ de saisie par une liste de choix, la liste présentant le nom des étudiants, ordonnées par ordre alphabétique.

Note : se référer à l'annexe 2 du document, méthode **`collection_select`**, en se limitant aux quatre derniers paramètres, le premier (le type de ressource) étant omis étant donné qu'elle s'inscrit dans un formulaire « **`form_for`** » qui spécifie déjà le type de la ressource en question (@portable):

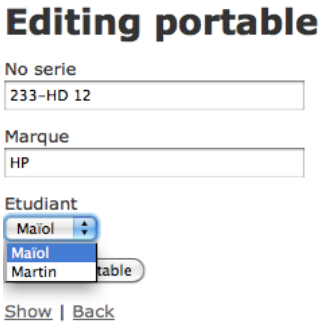
```
<%= form_for(@portable) do |f| %>

  [..]
```

```
<div class="field">
      <%= f.label :marque %><br />
      <%= f.text_field :marque %>
 </div>
 <div class="field">
      <%= f.label :Etudiant %><br />
      <%= f.collection_select .. à compléter .. %>
 </div>
 <div class="actions">
      <%= f.submit %>
 </div>
<% end %>
```

La soumission du formulaire entraînera alors l'exécution de l'action `update` avec le paramètre :

```
@params =
      {id => 3,                          Primary key de la ressource portable
      "portable"=>{"no_serie"=>"PC123",   Données saisies dans le formulaire
                   "marque"=>"DELL",
                   "etudiant_id"=>2}
      }
```

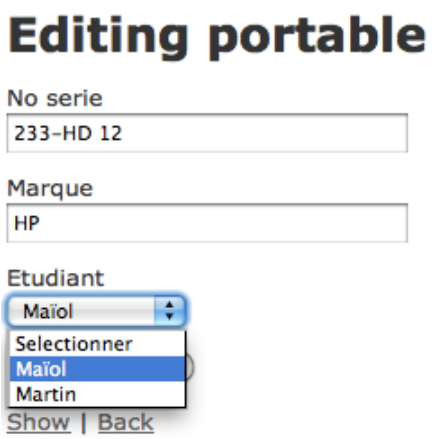


☺ ***Solution : Voir étape 4***

⇨ AMELIORATION NO4 - DU COTE PORTABLES

Dans une deuxième phase, faire en sorte que la liste déroulante contienne en première place le texte « `Selectionner` », indiquant que l'utilisateur doit sélectionner un élément dans la liste. Utiliser l'option « `include_blank` » de la méthode `collection_select`.

De plus, dans le cas d'une création de portable (new), la liste des étudiants apparaissant dans le combo-box devra être limitée à la liste de tous les étudiants **qui n'ont pas encore de portable attribué**. En cas de mise à jour du portable, la liste comprendra la totalité des étudiants.



Design MVC

Votre solution consistera à créer une méthode de classe `etudiants_sans_portable` dans le modèle `Etudiant`:

```
def self.etudiants_sans_portable
[..]
end
```

Cette méthode retournera un tableau contenant la liste des étudiants qui n'ont pas encore de portable.

Règles de base pour un bon design MVC

- **Un contrôleur doit rester le plus court possible !**
- **Tous les traitements doivent être opérés dans la partie modèle !**

Le code des méthodes `edit` et `new` ressemblera donc à :

```
def new
      @portable = Portable.new
      @etudiants = Etudiant.etudiants_sans_portables
[..]

def edit
    @etudiants=Etudiant.all
end
```

→ Requête SQL retournant les étudiants qui n'ont pas encore de portable attribué:

```
SELECT * FROM etudiants
WHERE etudiants.id NOT IN
          (  SELECT distinct etudiant_id
             FROM portables
             WHERE etudiant_id IS NOT NULL)
ORDER BY etudiants.nom
```

☺ ***Solution avec une requête SQL : Voir étape 5***

☺ ***Solution sans requête SQL : Voir étape 5-bis***

⇨ AMELIORATION NO5 – TESTER QUE LE MEME PORTABLE N'EST PAS ATTRIBUE PLUSIEURS FOIS

Dans la version actuelle, il est possible d'allouer différents portables au même étudiant.. (en effet, dans l'édition d'un portable, plutôt que d'avoir restreint les étudiants à ceux qui n'avaient pas de portable, on peut sélectionner n'importe quel étudiant..)

Faire en sorte qu'une validation soit opérée au moment de la sauvegarde: si un portable x est affecté à un étudiant a qui un portable a déjà été alloué, générer une erreur avec le message « Cet étudiant a déjà un portable ». Utiliser pour ce faire la clause « **`validate`** » (Se référer à l'annexe du document « Validation de saisie »)

☺ ***Solution: Voir étape 6***

⇨ AMELIORATION NO6 – AJOUTER UN MENU GENERAL

Faire en sorte qu'apparaissent toujours deux hyperliens dirigeant l'utilisateur sur la gestion des portables, respectivement sur la gestion des étudiants.

Faire en sorte également qu'au départ de l'applicatif l'on soit dirigé vers l'index des portables.

☺ ***Solution : Voir étape 7***

7 ANNEXE 1 : VALIDATION DE SAISIE

On peut introduire des « clauses » demandant à ce que certaines vérifications soient opérées pour certains champs au moment de la création ou de leur modification, avant que ne soit introduite la modification dans la base de données.

Ces clauses doivent être placées dans le modèle lui-même :

```
class Etudiant < ActiveRecord::Base
```

Champ non "null"

```
validates_presence_of :nom, :prenom, etc..
```

Pour spécifier les messages qui seront associés à l'erreur, écrivez une validation spécifique pour chacun des champs.

```
validates_presence_of :nom, :message => "Saisir le nom !! "
```

Champ de type numérique

Pour contrôler que le texte saisi peut être converti en nombre

```
validates_numericality_of :unChamp, :autrechamp, etc..
```

Unicité de la valeur

Pour contrôler que la valeur saisie est unique dans la colonne.

```
validates_uniqueness_of :unChamp, :autrechamp, etc..
```

validation d'un format

Pour contrôler un format selon une expression régulière.

```
validates_format_of :unChamp,
        :with => %r{\.(gig|jpg|png)$}i,
        :message => "doit être l'URL d'une image GIF, PNG,
        etc.. "
```

Autre contrôle: `validate()`

 Pour effectuer des contrôles plus complexes, on peut indiquer les tests effectuer en utilisant la clause **validate** :

```
validate :unTest, :unAutreTest
```

Puis, plus loin, déclarer les méthodes correspondantes :

```
private
def unTest
      if price.nil? || price <0.01
            errors.add( :price, "doit être > 00.1")
      end
end
def unAutreTest
      if no.nil? || (no < 100 || no > 1000)
            errors.add( :no, "doit être compris entre 100 et 1000")
      end
end
```

8 ANNEXE 2 : LE COMBO-BOX « COLLECTION-SELECT »

L'assistant `collection_select()` permet de travailler directement avec une collection d'éléments dont chaque membre posséderait aux moins deux attributs, le premier indiquant la valeur retournée en cas de sélection, et le deuxième le texte d'affichage de l'option.

Voici un exemple:

```
<%=
@users = User.find(:all, :order => "name")
collection_select(:user, :name, @users, :id, :name,
      {:include_blank => 'Please select'})
```

Explication sur les paramètres :

`:user` ➔ modèle utilisé

`:name` ➔ nom de l'attribut concerné (pour le retour des paramètres dans params)

`@users` ➔ la collection d'éléments

`:id` ➔ pour spécifier la valeur qui sera retournée à la soumission du formulaire

`:name` ➔ pour spécifier l'attribut qui sera affiché dans le combo-box

```
{:include_blank => 'Please select'})
```

➔ Un paramètre optionnel permettant de ne rien sélectionner du tout (retournera la valeur nil)

```
%>
```

La première instruction opère une recherche dans la base de donnés. La variable `@users` est une collection d'éléments possédant chacun des attributs, dont `:id` et `:name` qui seront utilisés dans l'assistant pour spécifier la valeur retournée (`:id`) ainsi que la valeur d'affichage (`:name`) pour chaque utilisateur apparaissant dans la liste d'options.

Dans le contexte d'un formulaire « `form_for` », la méthode sera invoquée en envoyant le message au formulaire `f`, et le premier paramètre sera omis.

➔ **`f`**`.collection_select(:name, @users, :id, :name)`