Haute Ecole LÉONARD DE VINCI

INSTITUT PAUL LAMBIN

Section Informatique

Université Catholique de Louvain

Analyse et développement d'une application web responsive destinée à enrichir la communication professeur-étudiant durant un cours

Mémoire présenté par

YEMNI, DAOUD

en vue de l'obtention du diplôme de bachelier en Informatique de gestion

Année académique 2014-2015

Remerciements

Je tiens à remercier :

- Le corps professoral de l'Institut Paul Lambin

Pour leur enseignement de qualité qui m'a permis d'avoir de nombreuses bonnes connaissances dans le monde informatique.

- L'équipe de l'ECAM

Malgré les moments difficiles, ils ont toujours été là à me conseiller, à m'aider que ce soit dans le cadre du stage ou dans mes choix personnels pour mon avenir.

- Mes parents

Pour le soutien qu'ils m'ont offert durant mon parcours scolaire et me rappeler mes devoirs en tant qu'étudiant et stagiaire.

- Christophe Damas

Mon superviseur de stage qui a pu m'aider dans les moments difficiles et a su me réorienté sur la bonne voie de manière très propre et pédagogique.

1 Table des matières

2 Introduction

Chaque année, le nombre d'étudiants en enseignement supérieur augmente constamment. Cette augmentation donne de plus en plus de difficulté aux enseignants à gérer leurs cours et les aides possible envers les étudiants.

C'est pourquoi les professeurs désirent une assistance informatique pour répondre aux besoins de tous les étudiants, à aider les étudiants dans la difficulté, à les soutenir et questionner les étudiants pendant un cours.

Lorsqu'un professeur débute un de ses cours. Il se connecte à l'application et décide de créer un topic lié à ce cours. Les étudiants se joignent à ce topic, et peuvent à tout moment du cours y placer des commentaires qui soient liées au cours. A la fin ou pendant le cours, le professeur peut décider de soumettre un sondage aux étudiants, leur laisser le temps d'y répondre et de clore le sondage. Il peut aussi, quand il le désire, consulter les commentaires et d'y répondre oralement à son cours et/ou de valider les commentaires qu'il estime correct ou intéressant.

L'application devra donner une interface ergonomique, rapide et simple d'utilisation que ce soit pour un professeur ou un étudiant. Elle devra être en temps réel afin d'éviter de gaspiller du temps. Elle permettra aux professeurs de pouvoir créer des topics liés aux cours, de pouvoir créer ou relancer des questions/sondages, de pouvoir afficher sous forme de statistique les réponses d'une question/sondage, de pouvoir consulter les questions ou commentaires des étudiants. L'application permettra aux étudiants de pouvoir assister à un topic et de pouvoir répondre aux questions ou sondages et de mettre des commentaires seulement quand le professeur désire que cela soit possible.

3 Contexte de travail

3.1 Société

Ecole Centrale des Arts et Métiers, ECAM, est une école d'enseignement supérieur. Elle forme les étudiants en tant que rang d'ingénieur industriel selon différentes spécialités :

- Automatisation
- Construction
- Électromécanique
- Électronique
- Géomètre
- Informatique

3.2 Environnement informatique

L'environnement de travail utilisé est principalement Linux et Windows. Pour le développement du projet de stage, nous avons utilisé comme outil logiciel :

- puTTy
- Vim
- Sublime Text
- Rails
- Faye
- Navigateurs

Les langages utilisés sont :

- Ruby
- Slim (HTML condensé)
- HTML
- Coffeescript (Javascript condensé)
- AJAX
- Markdown.

3.3 L'équipe

L'équipe informatique est constitué de 3 personnes, dont 2 sont des professeurs. Le projet du stage était un projet solo. J'avais la tâche de développer l'application seul mais avec le soutien des informaticiens en cas de besoin.
L'interaction avec les informaticiens se passait bien avec la majorité de l'équipe.

4 Calendrier

Calendrier du stage par semaine :

Semaine 1 *16 au 20 février 2015*	• Installation du système et de l'environnement de travail • Début de la formation sur Rails et Ruby via tutoriel et vidéo en ligne.
Semaine 2 *23 au 27 février 2015*	• Apprentissage des bases, modèle MVC de Rails via la réalisation d'un mini-projet
Semaine 3 *2 au 6 mars 2015*	• Fin de la formation • Mise en place de la base et des plugins (obligatoires) pour le projet du stage • Rédaction du cahier de charge abstrait • Mise en place des bases : formulaire de création d'un topic, sondage/question et commentaire
Semaine 4 *9 au 13 mars 2015*	• Validation du cahier de charge abstrait • Implémentation des cas d'utilisation • Mise en place des UC : « [PROF] Créer un sondage », « [PROF] Créer un topic », « Créer un commentaire », « [ETUD] Pouvoir répondre à un sondage », « Mettre en favori » et « [PROF] Valider un commentaire » • Mise en place d'une nouvelle fonctionnalité : « Mise en place d'un label individuel pour un topic »
Semaine 5 *16 au 20 mars 2015*	• Avancement du développement du projet & mise en place d'un design correct • Correction des bugs & optimisation du code • Installation du système de Login ECAM • Réunion afin de voir l'avancement du projet + DEMO • Ajout de nouvelle fonctionnalité : « Editeur texte avancé », « U.D. sondage/question », un étudiant peut voir sa réponse
Semaine 6 *23 au 27 mars 2015*	• Implémentation d'un système de recherche avancé via des labels • Amélioration du design
Semaine 7 *30 mars au 3 avril 2015*	• Implémentation du second serveur de live-updates (Faye) • Implémentation d'un système de « prévisualisation »
Semaine 8 *6 au 10 avril 2015*	• Réunion : ajout d'une nouvelle fonctionnalité « Promouvoir un sondage » • Optimisation du code back-end • Amélioration du design
Semaine 9 *13 au 17 avril 2015*	• Optimisation du code back-end • Correction des bugs du service live-updates • Correction de bugs d'affichage ou d'interaction • Réunion + visite du promoteur
Semaine 10 *20 au 24 avril 2015*	• Test manuelle des interactions de l'application • Ajout de sécurité • Correction de bug d'interaction • Amélioration du système de recherche (+ général, + performant, ajout de filtre)
Semaine 11 *27 avril au 1 mai 2015*	• Mise à jour des vues pour qu'elles soient responsives (adaptable que ce soit sur mobile, tablette ou pc)
Semaine 12 *4 au 8 mai 2015*	• Optimisation des vues • Optimisation des requêtes du DAO • Ajout de la pagination dans le système de recherche • Début de la mise en place des tests fonctionnels

Semaine 13 *11 au 15 mai 2015*	• Installation des plugins nécessaires aux tests • Mise en place des tests fonctionnels • Correction des bugs (découvert par les tests)
Semaine 14 *18 au 22 mai 2015*	• Finition des quelques tests (tous les tests n'ont pu être faits) • Retrait des dépendances de l'ECAM • Ajout d'un système d'inscription/connexion

5 Technologies

5.1 Ruby on Rails

5.1.1 Qu'est-ce que Ruby on Rails ?

Ruby on Rails est un Framework web libre écrit en Ruby.

Celui-ci permet de concevoir une application web de manière rapide, simple et complète. Cependant, pour que cette simplicité et rapidité soient vraies, il faut respecter les conventions, syntaxes et structure du Framework. Il répartit les différents services afin que les développeurs puissent se concentrer uniquement sur le développement sans perdre du temps dans la configuration.

Rails se base sur 3 principes fondamentales :

- D.R.Y. (Don't Repeat Yourself) :
  Pattern visant la réduction de duplication de code, évitant les copier/coller.

- ReST (Representional State Transfert) :
  Pattern visant l'utilisation d'identificateur de ressources dans les URLs

- CoC (Convention over Configuration) :
  Pratique visant à inciter aux développeurs à être conventionnel. C'est-à-dire, par exemple, s'il existe un modèle qui s'appelle « Vente », il est alors, de convention, logique qu'une table de données se nommera « ventes ». En respectant cette convention, les développeurs perdront moins de temps à configurer le Framework.

5.1.2 Qu'est-ce que Ruby ?

Énoncé au début, RoR est basé sur le langage Ruby.

Ruby est un langage de script orienté objet créé en 1993 au Japon par Yukihiro Matsumoto. Ce langage est destiné principalement au développeur web. En effet, il a été conçu afin que le code soit simple, efficace et élégant que ce soit pour celui qui écrit ou qui lit.

Il simplifie l'écriture de code en permettant au développeur de pouvoir faire abstraction de chose logique. Par exemple, vous pouvez faire abstraction des « {} » ou des « () » ou du « ; » ou du « return », etc.... Il s'agit aussi d'un langage non typé, le type est défini lors de la compilation.

Exemple :
Voici une méthode « calculer » qui additionne « a » et « b ».

- Java :

```java
public static int calculer(int a, int b){
    return a + b;
}
```

- Ruby :

```ruby
def calculer(a, b)
    a + b
```

Le langage simplifie la manière d'écrire du code.

Même lors d'un appel de fonction, vous pouvez faire abstraction des « () » par exemple.

- Java :

```java
public class Math {

    public static void main {
        calculer(10, 2);
    }

    public static int calculer(int a, int b){
        return a + b;
    }

}
```

- Ruby :

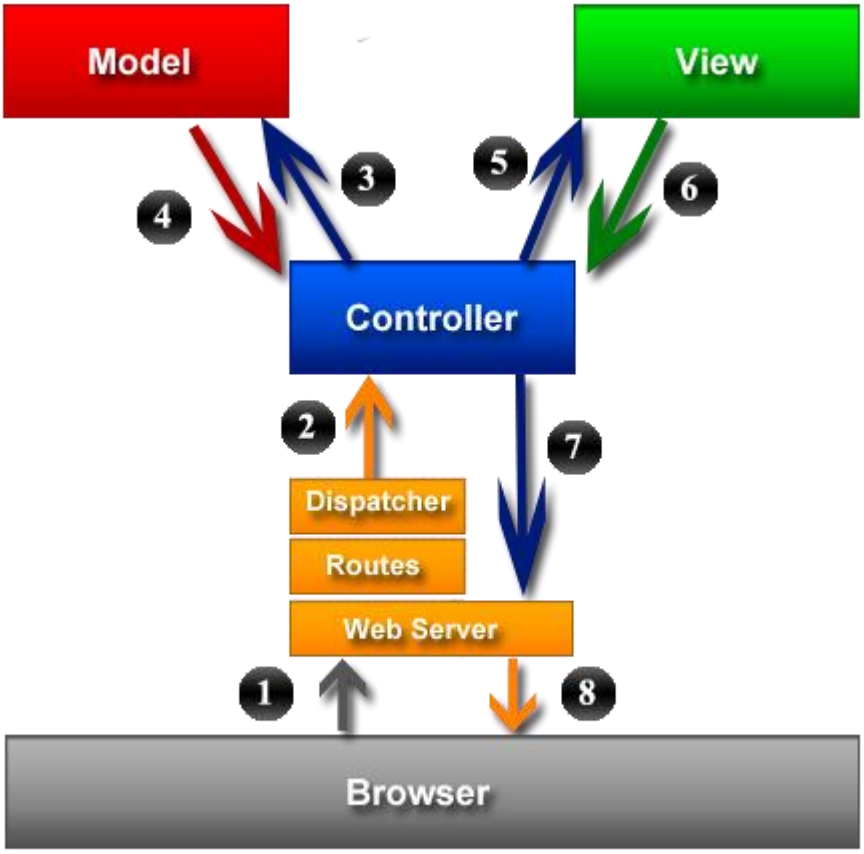
```ruby
class Math

    calculer 10, 2

    def calculer(a, b)
        a + b
```

Vous pouvez faire abstraction des « {} » mais pour cela vous devez utiliser l'indentation. Dans l'exemple, le code « a + b » est indenté car elle fait partie de la méthode 'calculer', qui elle-même fait partie de la classe « Math ».

Le langage implique certaine convention pour que cela reste toujours compréhensible et simple à utiliser.

5.1.3 Architecture

Rails nous impose le modèle MVC :



L'architecture du Framework Rails permet de développer une application assez rapidement, car plusieurs choses sont générées pour aider les développeurs. Tout d'abord, nous allons comprendre l'architecture de Rails en déchiffrant le diagramme ci-dessus.

Lorsqu'un utilisateur veut accéder à notre application web via son Browser (1). La requête est d'abord envoyer au serveur web. Le serveur, via l'url, va lire la route introduite par l'utilisateur.

« *Qu'est-ce qu'une route dans le Framework Rails* ? »

Une route est un chemin d'accès vers une ressource.
Exemple : Un utilisateur accède à l'URL suivant : « http://example.com/users/sign_in », nous avons le nom de DNS du site « example.com/ » et nous avons la route « users/sign_in ». Pour gérer les routes, il y a un fichier dans notre projet rails « config/routes.rb ». Nous pouvons ainsi configurer nos routes comme nous le voulons.

Équation 1 Exemple de fichier routes.rb

```ruby
Rails.application.routes.draw do

  devise_scope :user do
  end

  devise_for :users, :controllers => { :omniauth_callbacks => "users/omniauth_callbacks", sessions: "users/sessions", registrations:

  resources :labels

  resources :topic_joined_users

  resources :vote_user_comments

  resources :favor_surveys

  resources :labels_topics

  resources :user_answers

  resources :answers

  resources :comments

  resources :surveys

  resources :topics

  resources :users

  root to: "users#redirect_home"
  post '/language' => "users#change_language", as: "change_language"

  # User
  get '/home' => 'users#home' , as: 'home'
  get '/search' => 'users#search', as: 'search'

  # Topic
  post 'topics/create' => 'topics#create', as: "new_create_topic"
  post 'topics/create/closed' => 'topics#create_closed', as: "create_topic_closed"
  post 'topics/:id/close' => 'topics#close', as: "close_topic"
  post 'topics/:id/join' => 'topics#join', as: "join_topic"
```

Voici un exemple de route généré via le fichier ‘routes.rb’ d’une application Rails :

Table 1 Routing

Helper Path / Url	HTTP Verb	Path Path Match	Controller#Action
topics_path	GET	/topics(.:format)	topics#index
	POST	/topics(.:format)	topics#create
new_topic_path	GET	/topics/new(.:format)	topics#new
edit_topic_path	GET	/topics/:id/edit(.:format)	topics#edit
topic_path	GET	/topics/:id(.:format)	topics#show
	PATCH	/topics/:id(.:format)	topics#update
	PUT	/topics/:id(.:format)	topics#update
	DELETE	/topics/:id(.:format)	topics#destroy
root_path	GET	/	topics#index

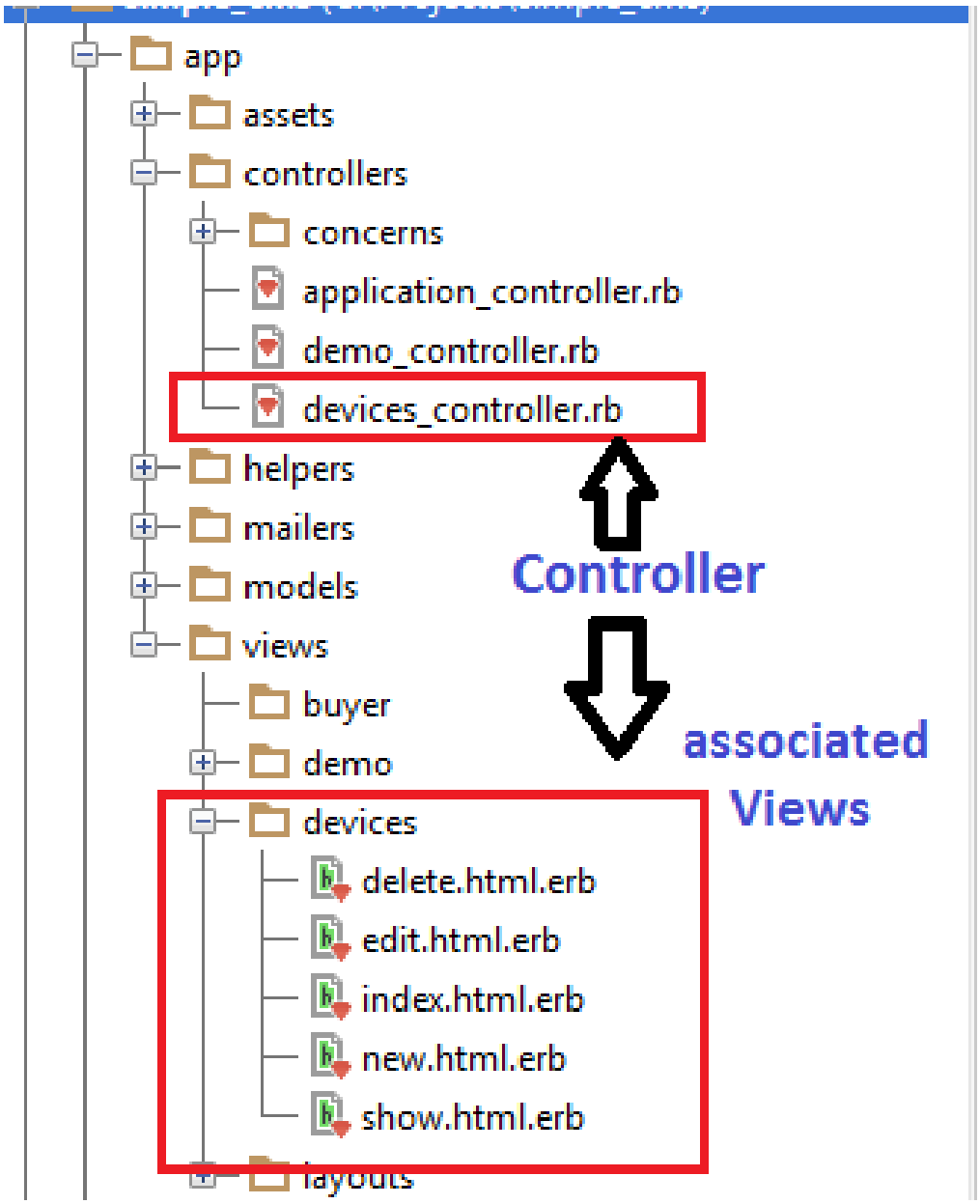
Une fois la route détectée, Rails va demander au Dispatcher de lui donner l’action et le contrôleur correspondant à la route.

Maintenant, l’application connait la route et la méthode/action et le contrôleur. Il va tout simplement appeler cette action (2).

Selon l’action/méthode appelé, il peut demander ou non des informations de la base de données. C’est là qu’entre en jeu, le modèle. Le modèle est le système qui gère la demande d’information de données de la DB, exactement comme un D.A.O. (3 & 4).

La méthode a, maintenant, les ressources nécessaires pour effectuer ces traitements et envoyez ses informations à la vue (5 & 6).

« *Comment la méthode du contrôleur va donner ses informations à une vue ?* »
Rails impose une nomenclature afin qu'il y ait un échange d'information entre la méthode du contrôleur et la vue correspondante. Le nom de la méthode doit être exactement le même que le nom du fichier représentant la vue. Ainsi en faisant cela, l'application sait que, par exemple, l'action « sign_up » du contrôleur « users » correspond au fichier « view/users/sign_up.html.erb ».
Voici un exemple, nous avons le « devices_controller » et ces vues associées sont dans le dossier « app/views/device ». Chaque fichier du dossier de la vue correspond à une méthode/action du « devices_controller »



Une fois la vue générée par les informations du contrôleur, le contrôleur va signaler au serveur web que la vue est prête et va lui envoyer celle-ci (7). Grâce à cela, le serveur web envoie la vue générée au Browser de l'utilisateur (8).

Système de plugins (gems)

En Rails, plein de plugins ou extensions existent pour améliorer notre application ou site web. Nous avons le système de plugin (gem). Cette fonctionnalité est l'une des plus importantes du Framework. En effet, grâce à cela, on peut accroître la rapidité de développement d'un projet en ajoutant directement des modules déjà implémentés dans notre application sans devoir le refaire de nous-même (D.R.Y.).

« Qui développe ses plugins ? »

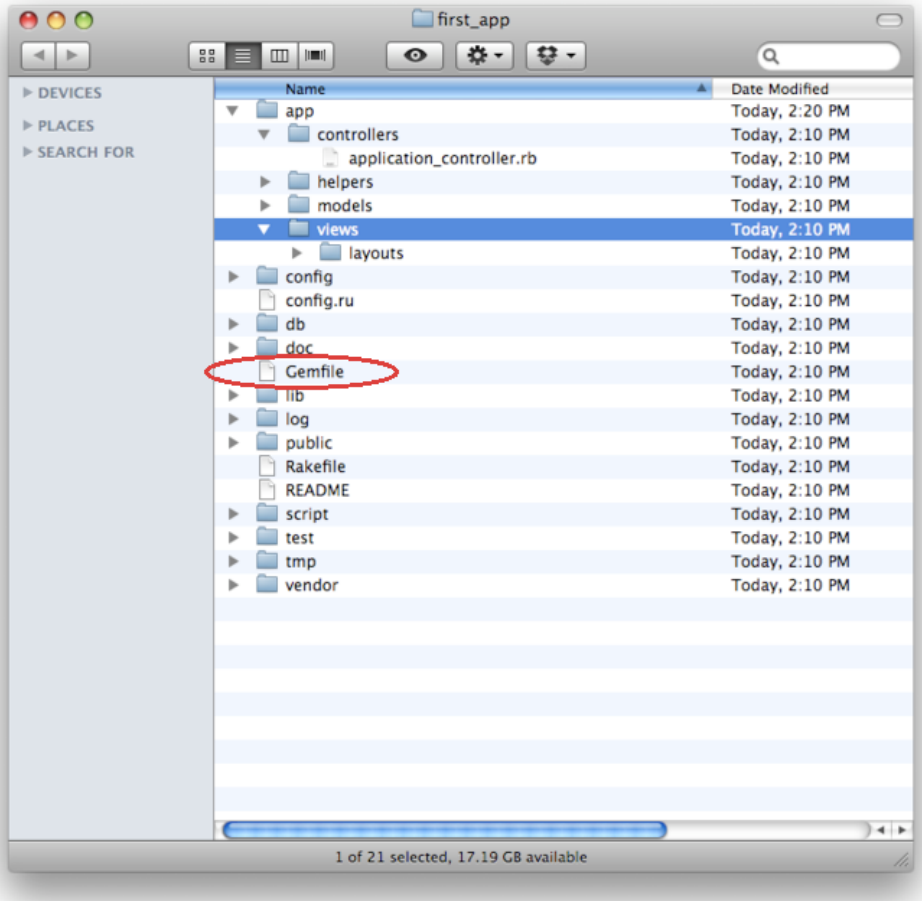
Ceux qui développent ses plug-ins sont tous simplement la communauté. En effet Rails laisse, à la communauté, la liberté de créer ses propres plugins.

« Les plugins, améliorent-ils notre application/site ou la manière de développer notre application/site ? »

Les plugins améliorent les 2. Il existe les plugins qui améliorent votre application ou site. Par exemple, il y a le plugin qui gère tous le système d'inscription et d'authentification qui existe déjà (DEVISE[1]). Au lieu de le réécrire, il suffit de l'installer et l'application ou site aura un système de login. Il existe plein de plugin à fonctionnalité différente.
Il existe aussi les plugins qui améliorent votre manière de développer. Par exemple, il existe un plugin qui permet de partager une variable créée depuis un contrôleur à notre script JavaScript (GON[2]).

Le système de plugin est simple à utiliser. Lorsqu'on génère le projet Rails, il installe une série de dossier et de fichier. Parmi ces fichiers, il y a un fichier « Gemfile » dans votre répertoire.

[1] Plugin de sign-up/in dans Rails (doc: https://github.com/plataformatec/devise)
[2] Plugin qui permet de donner accès de variables globales dans le JavaScript (doc: https://github.com/gazay/gon)



C'est à l'aide de ce fichier, que l'on peut installer les plugins à notre projet. Voici un exemple de contenu d'un fichier « Gemfile » :

La syntaxe de base, pour ajouter un plugin, est « gem 'nom_du_plugin' ». Il peut exister plusieurs versions à un plugin, c'est là que le 2e paramètre nous permet de fixer une contrainte sur la version du plugin que l'on veut.

Prenons par exemple « gem 'coffee-rails', '~> 4.0.0' ». Cette ligne dit que nous allons installer le plugin 'coffee-rails', uniquement la version 4.0.0 et que nous ne la mettrons pas à jour.

Nous pouvons utiliser d'autre signe d'égalité pour la version. Nous avons :

- « ~> X.X.X» qui signifie : exactement la version X.X.X
- « >= X.X.X » qui signifie : la dernière version possible mais supérieur à la version X.X.X

Une fois que nous ayons écrit les plugins nécessaires à notre développement. Il suffit de lancer une commande afin de les installer :

```
bundle install
```

Une fois vos plugins installés, nous pouvons les utiliser directement à notre projet.

Scaffolding

En RoR, il est possible de générer un ensemble de fichier pour une ressource : *scaffolding*.
Le *scaffolding* consiste à exécuter une ligne de commande qui permet de générer les fichiers de migration, les vues, le modèle et le contrôleur de la ressource demandé.

```
$rails generate ressourceName attr1:type attr2:type
```

Figure 1 scaffolding

Un fichier de migration [3]est un fichier qui va permettre de générer une table dans la base de données

```
1  class CreateUsers < ActiveRecord::Migration
2    def change
3      create_table :users do |t|
4        t.string :uid
5        t.string :identity_url
6
7        t.timestamps null: false
8      end
9    end
10 end
```

Figure 2 Exemple de fichier de migration

Une fois que nous avons bien généré nos fichiers pour les ressources désirées, on peut créer la base de données à l'aide de cette commande, qui se basera sur les fichiers de migration :

```
$rake db:migrate
```

Figure 3 Installation de la base de données

[3] Voir point 6.3.2 ce qu'est précisément un fichier de migration

5.2 FAYE

Faye est un système de publication et d'abonnement de message. En RoR, il n'y a pas de système de live-update, c'est-à-dire, un système qui met à jour les clients connectés lors d'une modification de données.

Faye offre ce service. L'installation, de ce service, est particulière. En effet, Faye est un serveur.

Tout d'abord, il faut créer un serveur Faye et il vous faut configurer votre fichier de config.

```
# config.ru

require 'faye'
Faye::WebSocket.load_adapter('thin')

app = Faye::RackAdapter.new(:mount => '/faye', :timeout => 25)

run app
```

Figure 4 Fichier de configuration d'un serveur faye

Quand votre configuration est terminée, vous pouvez lancer le serveur à l'aide de la commande : `$ thin start -R config.ru -p 4000` [4]

- -R, --rackup FILE : Option qui permet de définir le fichier de configuration du serveur
- -p, --port PORT : Option qui permet de définir le port d'accès (par défaut : 3000)

Maintenant que nous avons notre serveur Faye, nous allons expliquer comment fonctionne-t-il dans l'application.

Faye, comme expliqué plus haut, marche sous principe d'abonnement et de publication de message. Donc pour que nos vues se mettent à jour en temps réel, il faut qu'elle soit abonné pour que dès qu'il y a une modification de donnée et une publication, ces vues se mettent à jour.

Dans l'application de notre stage, nous avons fait en sorte que dès que l'utilisateur arrive sur notre application, il se connecte directement au serveur Faye :

```
270 $(document).ready ->
271     $.faye = new Faye.Client(gon.faye_url)
272     $.faye.disable('websocket')
273     $.faye.connect()
```

Figure 5 Connexion au serveur faye (CoffeeScript)

[4] Thin est un serveur web ecrit en Ruby. Lors du lancement du serveur, il se base sur un fichier de configuration pour le serveur.

Maintenant que notre application, tant que l'utilisateur est dessus, est connectée à notre serveur Faye. On déclare qu'à chaque fois qu'il accède à une vue, on lui abonne cette vue à notre serveur Faye et on désabonne toutes les autres de cette utilisateur :

```
174  # Faye {{{
175  faye_event = ->
176      if $.sub
177          $.sub.cancel()
178      $.pathname = window.location.pathname
179      $.sub = $.faye.subscribe($.pathname, (data) ->
180          $('' + data)[0].click()
181          return
182      )
183  # }}}
```

Ici, nous avons fait une action simple, dès qu'une publication se produit, on clique sur un bouton qui va tout simplement rafraîchir la page.

Le client est connecté et abonné. Nous pouvons à présent commencer à mettre en place le système de publication, afin que la mise à jour en temps réel soit fonctionnelle.

Prenons comme exemple le scenario suivant :
J'ai un professeur A et un étudiant B connectés à l'application. Cela fait donc 2 connexions à mon serveur Faye. Ces 2 utilisateurs sont dans la page des sondages http://example.com/topics/1/surveys. Ceux-ci sont donc abonnés à cette vue. Le professeur A décide de créer un nouveau sondage et soumet son nouveau sondage, nous atterrissons dans cette méthode :

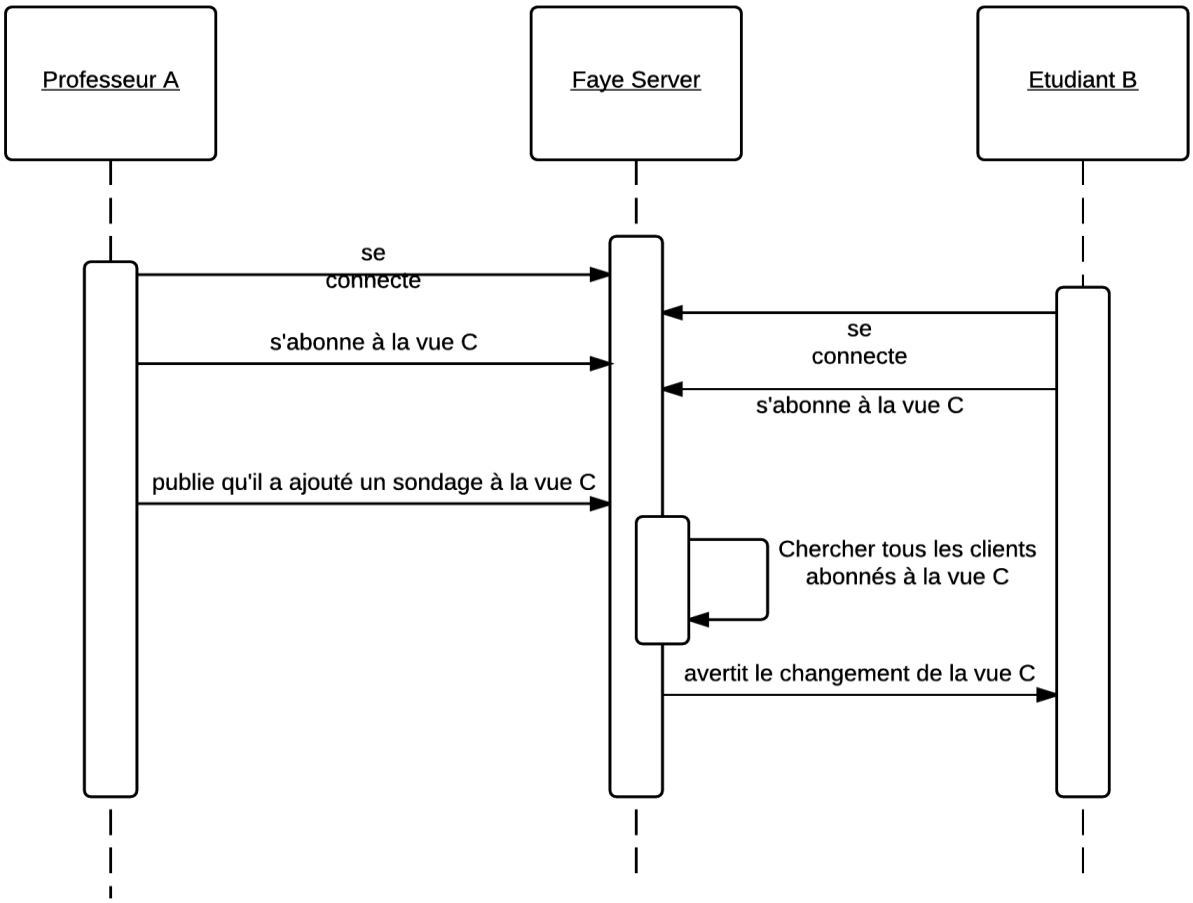
```
121   # POST /surveys
122   # POST /surveys.json
123   def create
124     @answers_attributes = params[:survey].delete(:answers)
125     @survey = Survey.new(survey_params)
126     @survey.user = current_user
127     @survey.status = @survey.topic.status
128     @topic = @survey.topic
129     respond_to do |format|
130       if @survey.save
131         if @survey.survey_type == I18n.t('survey.form.type_tf')
132           Answer.create( survey: @survey, title: I18n.t('survey.form.answer_true'))
133           Answer.create( survey: @survey, title: I18n.t('survey.form.answer_false'))
134         elsif @survey.survey_type == I18n.t('survey.form.type_multiple')
135           @answers_attributes.each do |a|
136             save_answer_qcm(a, @survey)
137           end
138         end
139         broadcast_msg_to_channel("#tab-surveys-topic", show_surveys_topic_path(@topic))
140         broadcast_msg_to_channel(".refresh-page", "/home")
141         format.js {render :layout => false}
142         format.html {redirect_to show_surveys_topic_path(@topic)}
143         # format.json { render :show, status: :created, location: @survey }
144       else
145         format.html { render :new }
146         # format.json { render json: @survey.errors, status: :unprocessable_entity }
147       end
148     end
```

On effectue tout le traitement de création du sondage, et on peut voir la méthode suivant :

```
broadcast_msg_to_channel("#tab-surveys-topic", show_surveys_topic_path(@topic))
broadcast_msg_to_channel(".refresh-page", "/home")
```

```
33    def broadcast_msg_to_channel(msg, channel)
34      message = {:channel => channel, :data => msg}
35      uri = URI.parse("#{Settings.faye_url}/faye")
36      Net::HTTP.post_form(uri, :message => message.to_json)
37    end
```

C'est grâce à cette méthode que l'on signale au serveur Faye qu'une publication a été effectuée. Nous ne publions pas réellement quelque chose, mais nous lui envoyons une requête pour lui déclarer qu'une publication a été produite. Le serveur Faye va donc récupérer tous les clients connectés où leur page est celle annoncé dans le 2e paramètre de la méthode et va appuyer sur le bouton déclaré dans le 1er paramètre.



6 Application

Contexte

TeachRoom est une application web, adaptée aussi pour mobile, live-update destiné à enrichir la communication entre étudiants et professeurs durant le cours. Elle permet d'aider un professeur à répondre ou questionner ces étudiants à son cours et de pouvoir lire les commentaires de ces étudiants. Le professeur doit pouvoir créer des topics liées à son cours, y poster des sondages ou répondre à des commentaires. Et un étudiant doit être capable de poster un commentaire à ce topic ou répondre aux sondages de celui-ci.

Définition

- Topic :

Un topic est un channel (comme sur un forum) où il est possible de poster des questions ou d'y répondre ou d'y mettre des commentaires.

- Sondage :

Un sondage est une question dédié à un ensemble de personne où on définit souvent à l'avance les réponses possibles et où on y traite de statistique.

Objectifs

L'application doit être simple à utiliser, rapide et clair. Ainsi, un professeur ne doit pas perdre de temps à chercher ce qu'il doit faire ou perdre son temps dans des formulaires inutiles.

Elle doit pouvoir automatiser des actions qui sont logiques.

Scenario

Voici un exemple d'utilisation :

Un professeur A arrive à l'école à 8h30 pour donner son cours B à l'auditoire. Le professeur, au début du cours, décide d'ouvrir un topic par rapport à son cours B (en 1 clic) et prévient les étudiants du local que le topic est ouvert. Le professeur donne son cours normalement.

Une fois vers la fin du cours, le professeur ré-ouvre l'application, retourne sur le topic qu'il a ouvert et dédié au cours. Il lit les commentaires et décide de répondre à certains oralement en classe. Il reste un peu de temps avant la fin du cours, alors le professeur décide de lancer un sondage et invite les étudiants à y répondre avant la fin du cours. Une fois la fin du cours, le professeur referme le sondage et le topic (s'il veut).

6.1 Situation

6.1.1 Avant

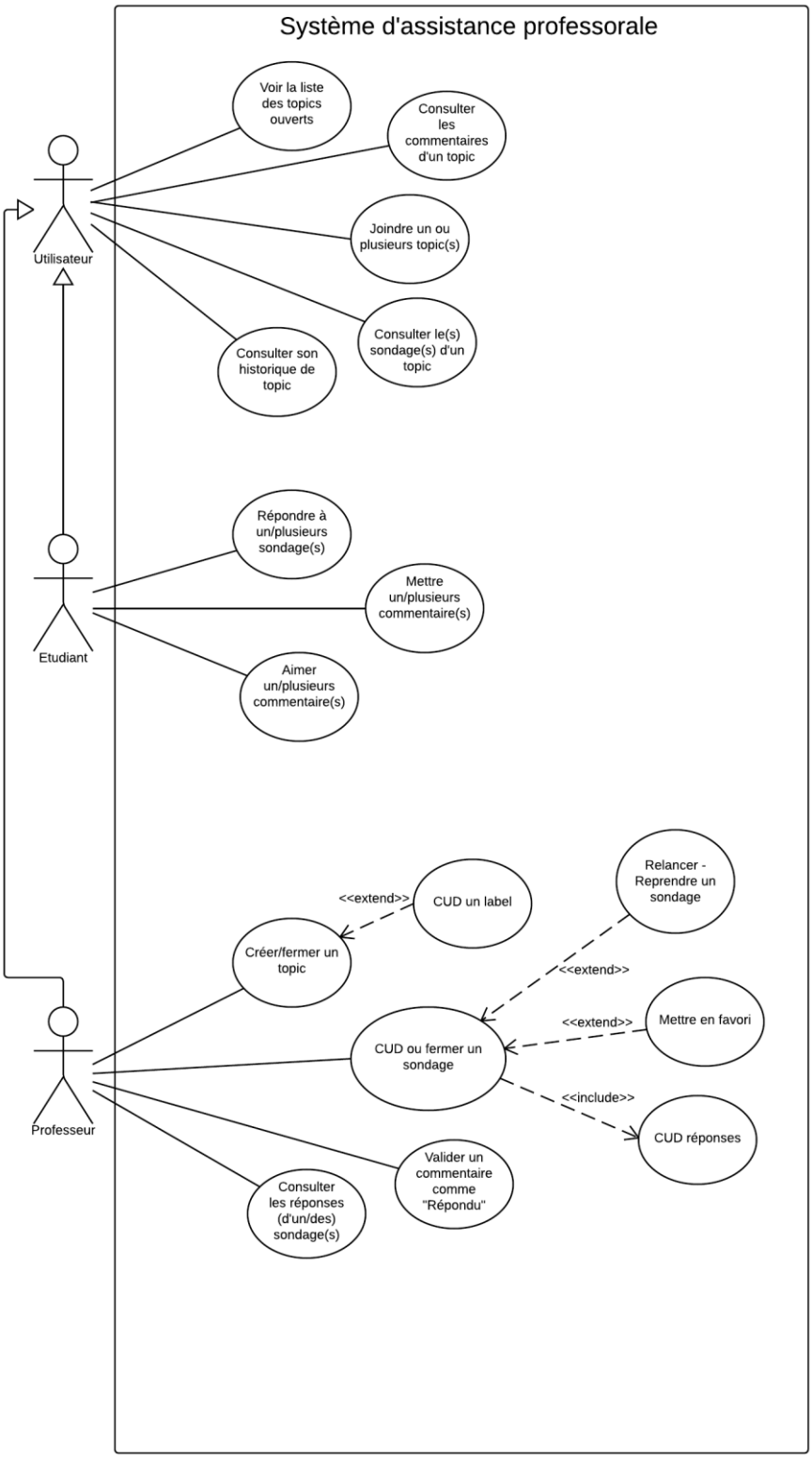
La Haute Ecole n'avait pas d'application d'aide à la communication aux professeurs durant leurs cours. L'idée fut venue d'un informaticien de l'établissement. L'informaticien est tombé sur une des vidéos des cours de Python de Google et il perçut une petite application qui gardait les commentaires et questions des étudiants du cours. Le professeur, de la vidéo, alla à la fin de son cours consulter leurs questions et commentaires et répondit à ceux-ci directement durant le cours. L'informaticien trouva l'idée judicieuse et décida de reprendre l'idée, de l'améliorer légèrement et de l'adapter au système de l'établissement.

Ainsi, c'est comme ceci qu'est né TeachRoom. Une application qui consiste à enrichir la communication entre professeur et étudiant durant un cours.

6.2 Analyse

6.2.1 Diagramme de cas d'utilisation

Dans le contexte, un utilisateur est un quidam qui est connecté à l'application. Il peut être soit un étudiant, soit un professeur.



a. Voir la liste des topics ouverts

Un utilisateur peut consulter la liste des topics ouverts courant. S'il s'agit d'un étudiant, il s'agit de sa page d'accueil. Tandis que le professeur peut accéder à cette vue via un onglet dans sa barre de navigation.

b. Créer ou fermer un topic

Un professeur peut créer un topic ou le fermer un topic.
Il peut ré-ouvrir un topic qu'il a déjà fermé mais il n'a le droit qu'à un seul topic ouvert en même temps.

c. Joindre un topic

Un utilisateur peut joindre un topic d'un professeur. S'il est professeur, il aura des droits particuliers pour ce topic et l'étudiant gardera ces droits de base.

d. CUD un label

Un utilisateur peut assigner ou supprimer un label à un topic. Chaque label est individuel, les autres utilisateurs ne peuvent pas voir ceux des autres. Cette action est facultative.

e. Consulter les sondages/questions d'un topic

Un utilisateur peut voir la liste des sondages/questions d'un topic. S'il est professeur, il pourra juste les consulter tandis qu'un étudiant peut répondre à ces sondages, s'ils sont ouvert.

f. Consulter les commentaires d'un topic

Un utilisateur peut voir les commentaires d'un topic. Deux actions sont possibles pour le professeur lorsqu'il consulte les commentaires. Il peut soit voir tous les commentaires (vue par défaut) soit voir les commentaires les mieux notées.

g. CUD un sondage

Le professeur a la possibilité de créer, modifier ou supprimer un sondage.

Lors de la création du sondage/question, le professeur choisit le type qu'il désire : QCM, Vrai/Faux ou question ouverte. Une fois, le choix effectué, il doit juste introduire la question. Seulement dans le cas d'un QCM, le professeur doit introduire les réponses qu'il désire sinon dans les autres cas, les réponses sont générées automatiquement.

Si un professeur désire modifier un de ses sondages/questions et ses réponses, il ne peut le faire seulement s'il n'y a pas eu de réponses des étudiants.

h. Répondre à un/plusieurs sondage(s)

Un étudiant peut répondre à un ou plusieurs sondage(s). Lorsque le sondage est clos, il pourra voir sa réponse et un petit diagramme du nombre de personne qui ont répondu par réponse.

i. Mettre en favori

Un professeur peut mettre en favori un sondage que ce soit le sien ou non. Cela lui permet de pouvoir relancer le sondage s'il ne lui appartient pas.
S'il relance le sondage, l'application va dupliquer la question et les réponses et réinitialiser les scores des réponses à 0.

j. CD un commentaire

Un étudiant peut créer un commentaire ou supprimer un de ses commentaires.
Tandis que le professeur peut créer un commentaire mais il peut supprimer n'importe quel commentaire d'un de ses topics.

k. Valider un commentaire

Un professeur peut valider un commentaire. S'il valide un commentaire, cela veut dire que le professeur a répondu à ce commentaire durant son cours oralement.

l. Aimer un ou plusieurs commentaire(s)

Un étudiant peut aimer un commentaire. Cette action impliquera que la note du commentaire augmentera. Plus la note du commentaire est haute, plus les chances, que le professeur le voit, sont grande, s'il les trie par ceux les mieux notés.

6.2.2 Diagramme de base de données

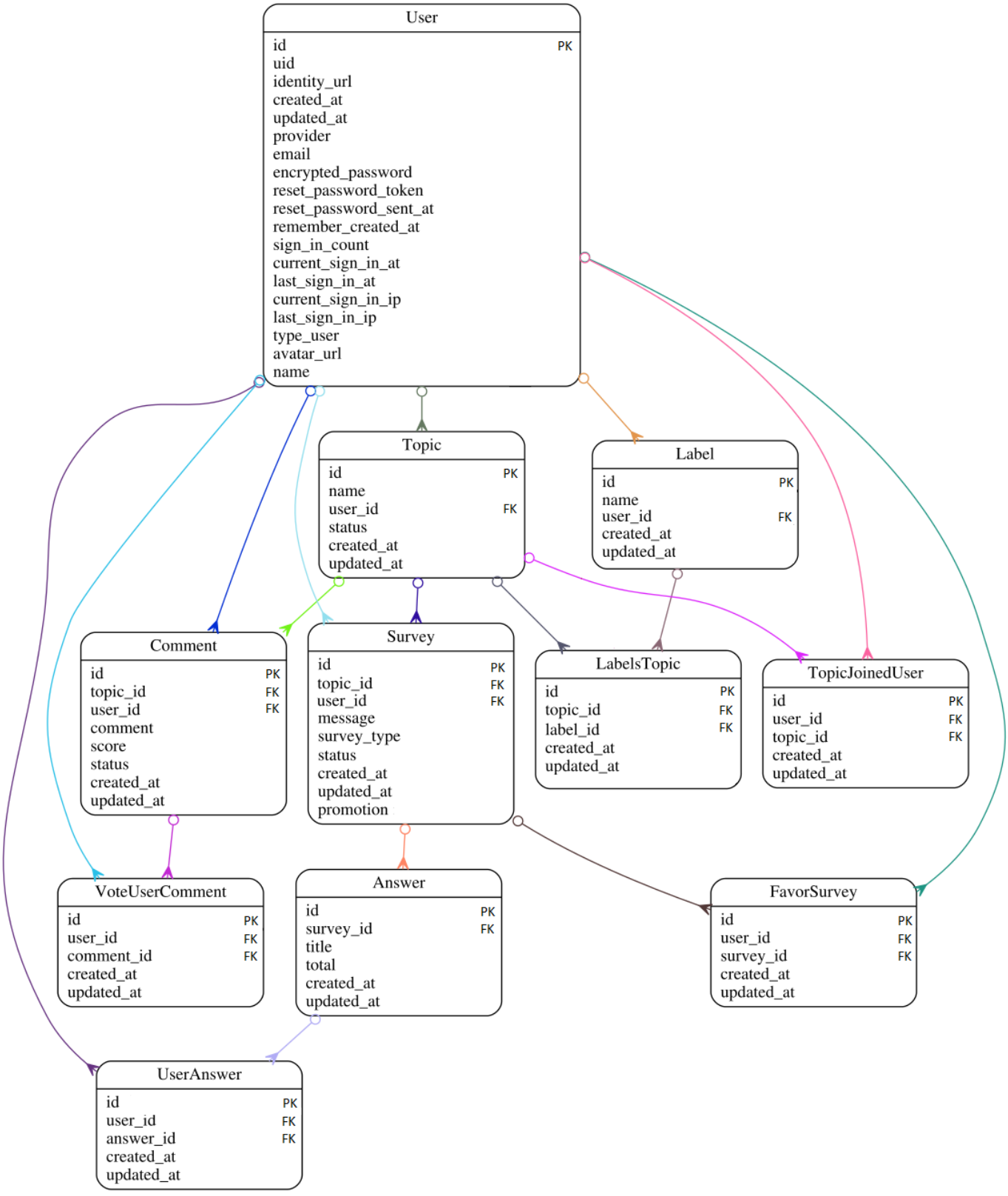
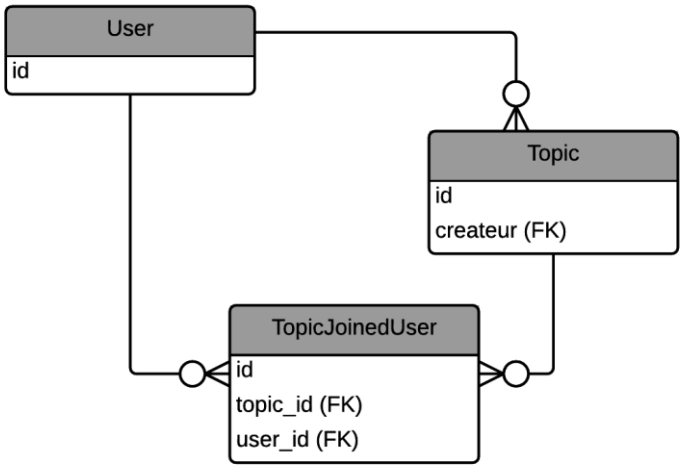


Diagramme généré à l'aide de la gem "Railroady"

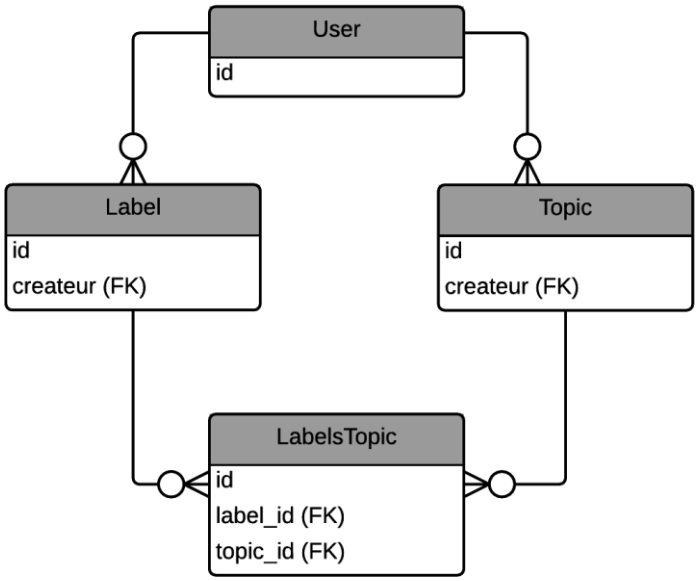
a. Relation : User – TopicJoinedUser – Topic



Dans notre application, nous avons parfois plein de professeur et d'étudiant qui passe par le même topic parce que ces ensembles d'étudiants et professeurs sont dans le même cours. Il nous fallait enregistrer les accès à un topic.

Cette relation permet d'enregistrer quel utilisateur est entré dans quel topic. Grâce à cette relation, on peut définir des statistiques (malheureusement non implémenté durant mon stage), c'est-à-dire : on peut savoir si un étudiant a été là à un cours ou de savoir quel était le professeur secondaire d'un cours, etc...

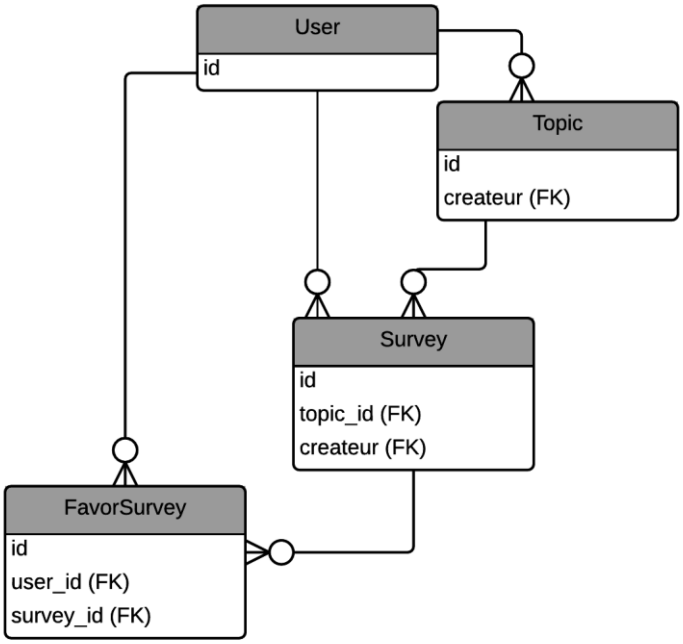
b. Relation : User – Topic – LabelsTopic – Label



Un utilisateur peut définir un label à un topic. Le problème est que le label est individuel, c'est-à-dire, l'utilisateur ne connait que ses propres labels et pas ceux des autres.

Le but du label est la possibilité de l'assigner à un topic, cela permet d'optimiser la recherche d'un topic. Cette relation permet d'enregistrer qu'un utilisateur a créé tel label et l'a assigné à tel topic. Ainsi, on préserve l'individualité sur le label.

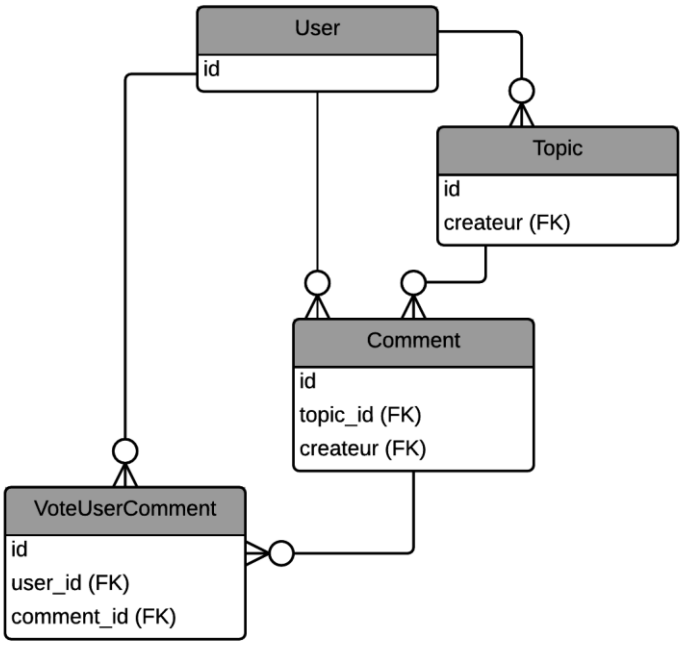
c. Relation: User – Topic – Survey – FavorSurvey



Ici, une relation a dû être ajoutée, la relation User – Survey. Vous me direz *« pourquoi rajoute-t-on une référence de user dans Survey alors que nous l'avons déjà dans topic ? »*. Nous avons dû l'ajouter tous simplement parce qu'il y a le cas où il y a plusieurs professeurs pour un même cours. Ainsi nous pouvons savoir qui est l'auteur d'un sondage même si l'auteur peut être un professeur secondaire d'un cours.

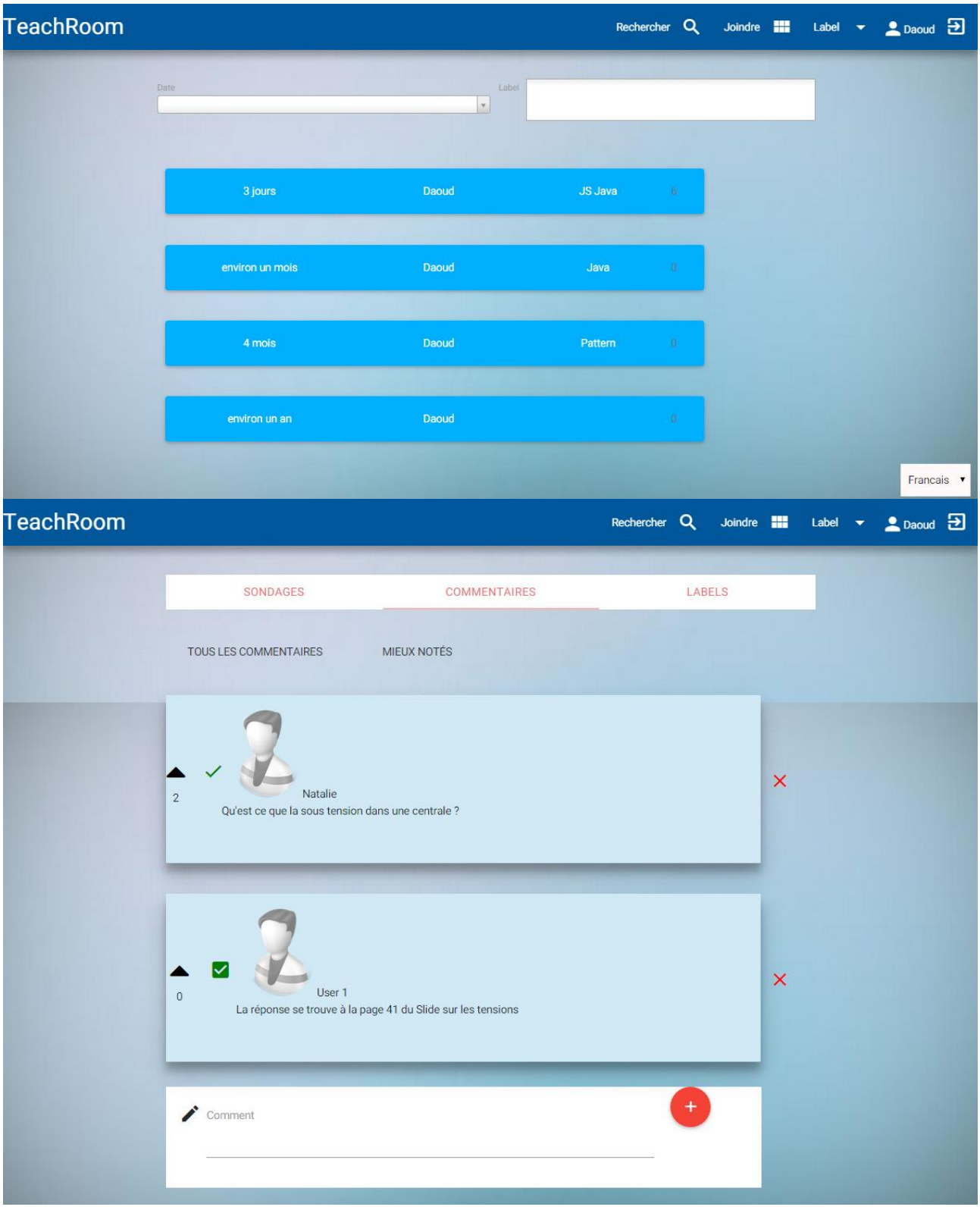
L'utilisateur a la possibilité de mettre en favori le sondage qu'il désire. La table intermédiaire « FavorSurvey » permet d'enregistrer quel utilisateur a mis quel sondage en favori. Ainsi, nous conservons le fait que chaque utilisateur a ses propres favoris. Nous avons la référence de l'utilisateur dans cette table, car il se peut que ce soit un professeur ou un étudiant qui mette en favori un sondage, qu'il n'en est ni l'auteur de celui-ci ni le créateur du topic.

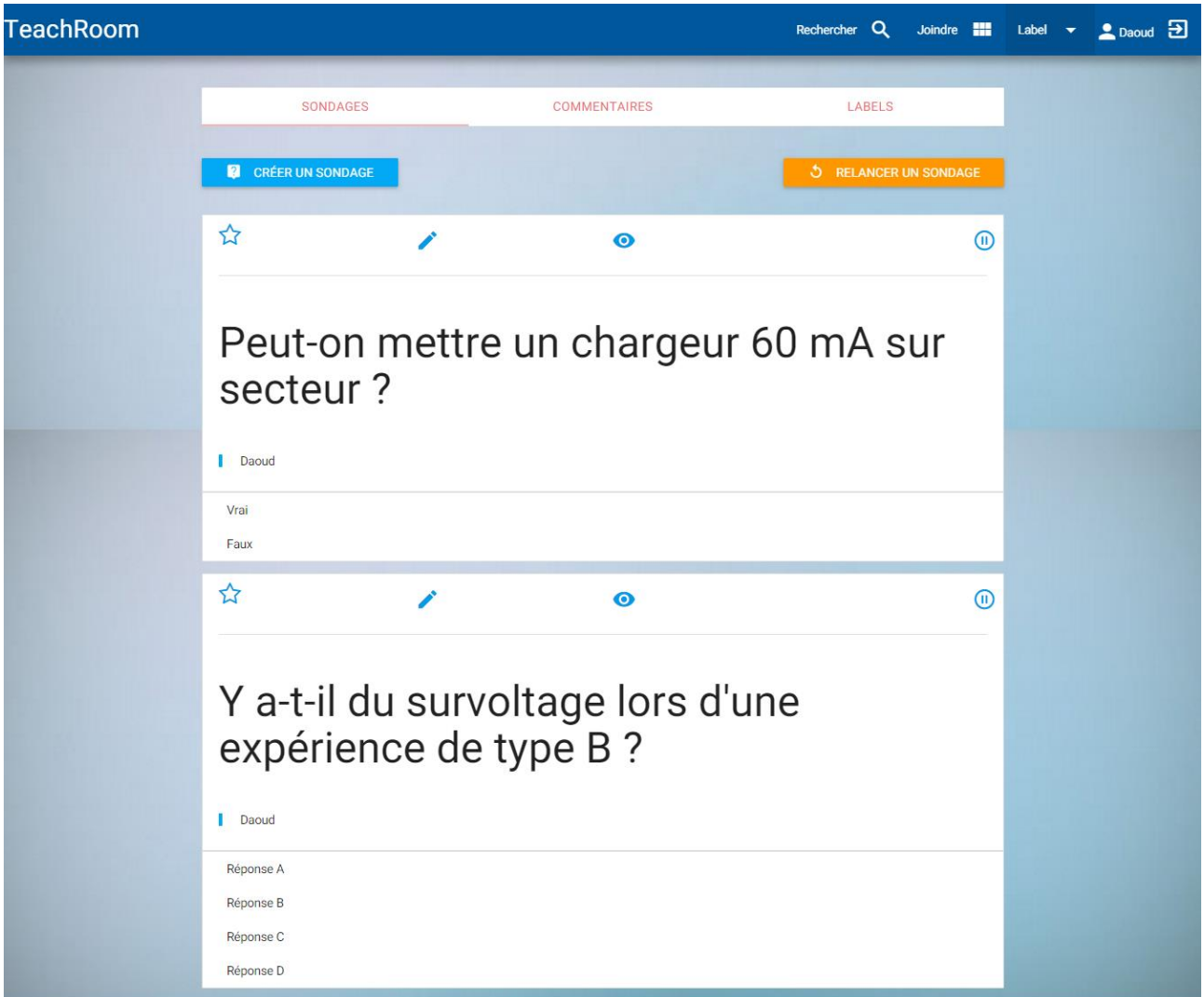
d. Relation User – Comment – VoteUserComment

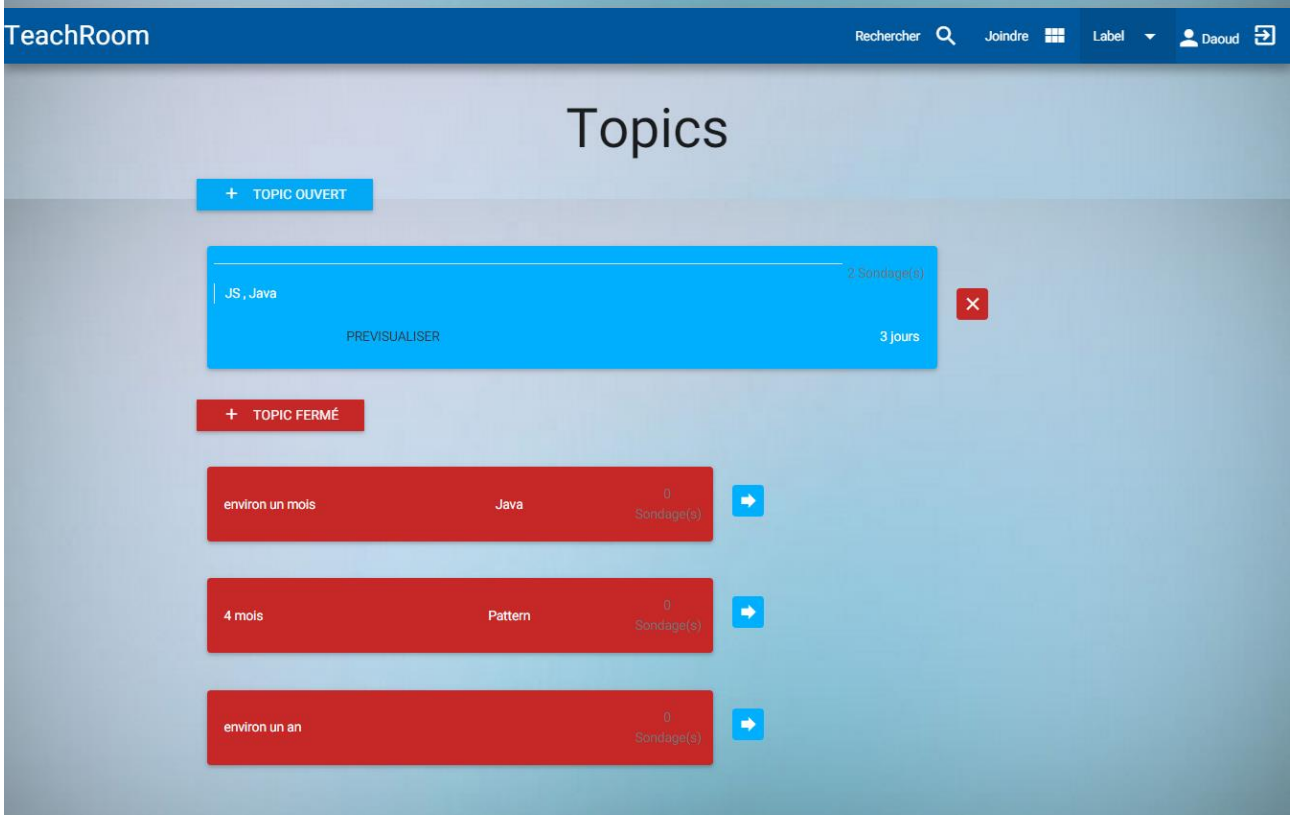


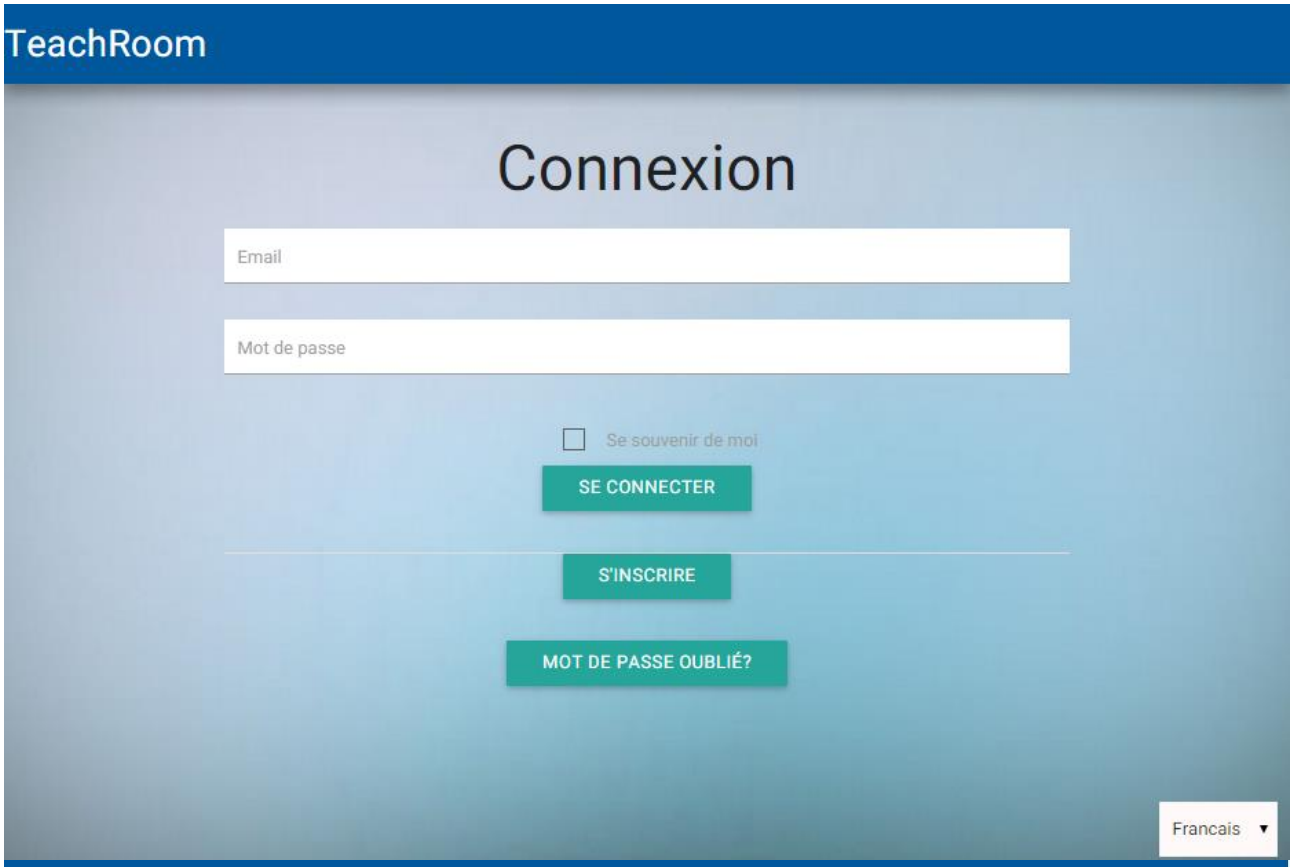
Un étudiant peut aimer le commentaire d'un autre étudiant. Mais dans notre application, chaque action est tracée. Grâce à la table intermédiaire « VoteUserComment », on peut enregistrer quel étudiant a aimé quelle commentaire. De la sorte, nous pouvons introduire des contraintes à l'utilisation de l'action « aimer en commentaire », c'est-à-dire, un étudiant n'a le droit que d'aimer 1 fois le commentaire d'un autre.

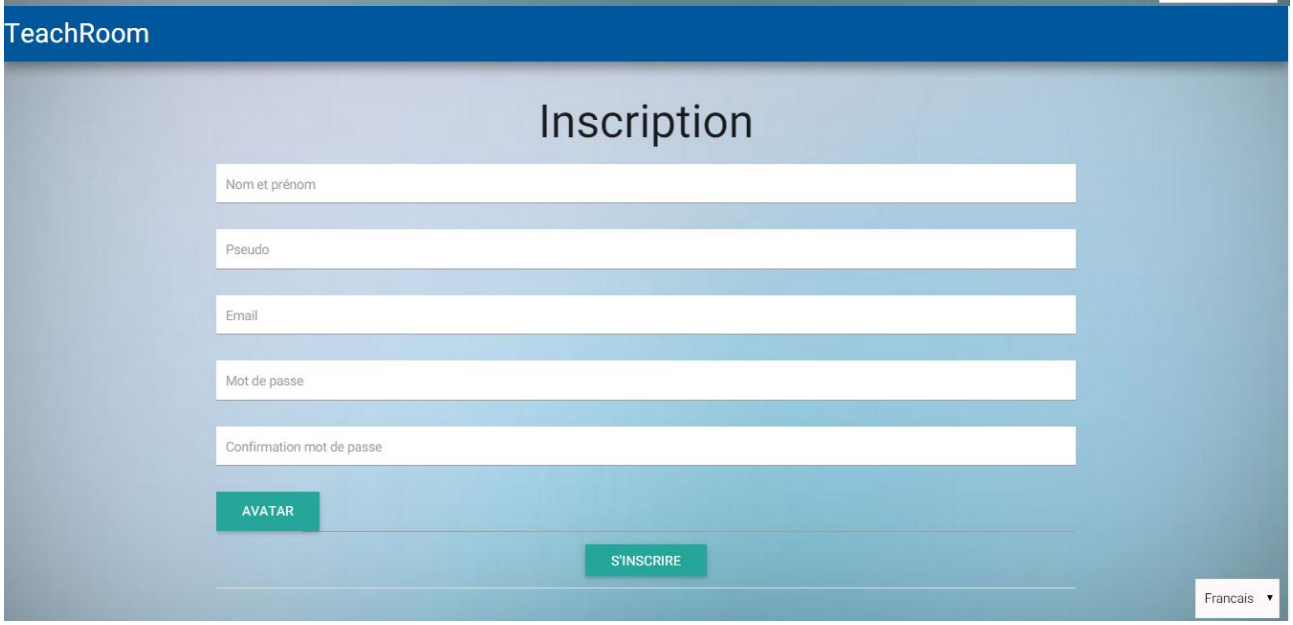
6.2.3 Prototype IHM




TeachRoom
Rechercher
Joindre
Label
Daoud
SONDAGES
COMMENTAIRES
LABELS
CRÉER UN SONDAGE
RELANCER UN SONDAGE
Peut-on mettre un chargeur 60 mA sur secteur ?
Daoud
Vrai
Faux
Y a-t-il du survoltage lors d'une expérience de type B ?
Daoud
Réponse A
Réponse B
Réponse C
Réponse D


TeachRoom
Rechercher
Joindre
Label
Daoud
Topics
TOPIC OUVERT
2 Sondage(s)
JS , Java
PREVISUALISER
3 jours
TOPIC FERMÉ
environ un mois
Java
0 Sondage(s)
4 mois
Pattern
0 Sondage(s)
environ un an
0 Sondage(s)


TeachRoom
Connexion
Email
Mot de passe
Se souvenir de moi
SE CONNECTER
S'INSCRIRE
MOT DE PASSE OUBLIÉ?
Francais


TeachRoom
Inscription
Nom et prénom
Pseudo
Email
Mot de passe
Confirmation mot de passe
AVATAR
S'INSCRIRE
Francais

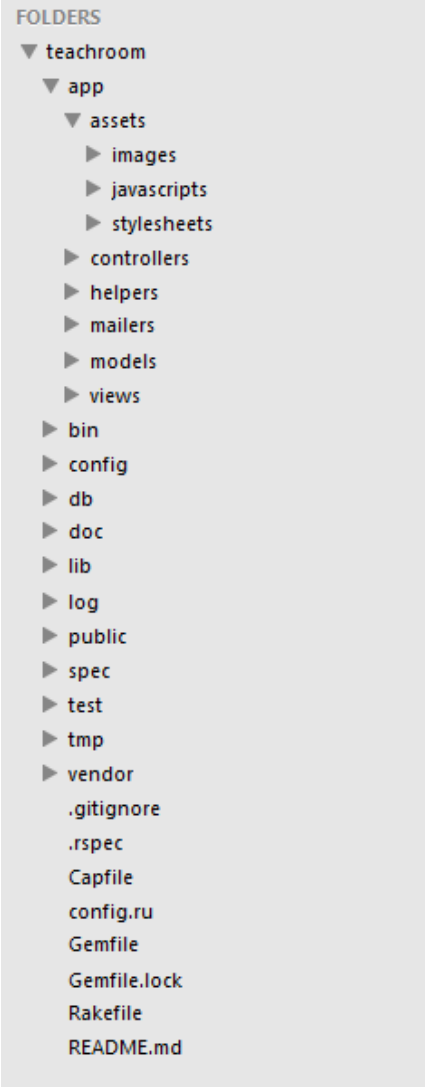
6.3 Implémentation

6.3.1 Création du projet

Pour pouvoir commencer le développement de notre application, nous devons d'abord générer le projet Rails du projet. Pour cela, nous devons exécuter une commande Rails qui va générer le dossier et la structure de notre projet.

```
rails new app_name
```

Structure



Certains dossiers et fichiers sont très importants dans le développement d'une application et vont être le plus souvent utilisés. Voici les dossiers ou fichiers les plus importants :

« app/ »	Un dossier contenant les fichiers concrets de l'application, tel que les modèles, les contrôleurs, les vues, le CSS et JavaScript.

« config/ »	Contient les fichiers de configurations importants pour l'application. Les fichiers de config pour chaque environnement (développement, production et test), le fichier de config des routes, …
« Gemfile »	Fichier permettant d'installer des plugins (gems) pour l'application

Maintenant que notre base du projet a été créée, nous pouvons commencer le développement de notre application.

6.3.2 Fichier de migration + Ressource

Notre projet est mis en place. Nous avons besoin, à présent, des ressources, de tables pour enregistrer nos informations relatives à notre application.

Qu'est-ce qu'une ressource ?

Une ressource est un objet que l'on a besoin pour des traitements CRUD (Create Read Update Delete) et que l'on peut y accéder via un URL. En Rails, une « ressource » est un terme utilisé pour spécifier une table, le modèle et le contrôleur lié à cette table et sa route d'accès.

Quelles ressources avons-nous besoin ?

Nous avons besoin de ressources mais lesquelles ? Grâce à l'analyse, nous avons besoin comme ressources de base de :

- Utilisateur
- Label
- Topic
- Sondage
- Commentaire

Pour les tables intermédiaires, nous n'allons pas les générer par *scaffolding* parce que pour certaine table nous n'avons pas besoin d'un contrôleur et d'une vue.

Fichier de migration

Un fichier de migration est un fichier comportant le code nécessaire pour créer la table + les attributs nécessaires à la ressource demandée.

Voici un exemple de fichier de migration généré pour la ressource Utilisateur.

```
1  class CreateUsers < ActiveRecord::Migration
2    def change
3      create_table :users do |t|
4        t.string :uid
5        t.string :identity_url
6
7        t.timestamps null: false
8      end
9    end
10 end
```

Ainsi nous pouvons générer des fichiers de migrations pour chacune des ressources nécessaires afin de mettre en place la base de données.

Après avoir générer tous les fichiers de migrations pour chacune des ressources, nous allons installer la base de données à l'aide de 'Rake'.

Rake est un logiciel de de gestionnaire de tâche comme le Make de Unix. Ici, Rake va être utilisé principalement pour tous les traitements concernant la base de données. Nous allons lancer Rake pour installer la base de données à partir des fichiers de migrations à l'aide de cette commande :

```
rake db:migrate
```

SeedFu est une gem qui permet d'insérer une série de données de départ ou fixe dans la base de données via un script sans devoir les insérer manuellement, un à un, via la console Rails.

```ruby
User.seed do |u|
  u.uid = "dao"
  u.name = "Daoud"
  u.email = "dao@teachroom.com"
  u.password = "daoud"
  u.type_user = "Teacher"
  u.avatar_url = "avatar-missing.png"
end

User.seed do |u|
  u.uid = "user1"
  u.name = "User 1"
  u.email = "user1@teachroom.com"
  u.password = "user1"
  u.type_user = "Teacher"
  u.avatar_url = "avatar-missing.png"
end
```

Figure 6 Exemple de fichier SeedFu

La bonne pratique serait que nous insérions nos codes d'insertions dans un fichier (en respectant la convention : 1 fichier par table).

Et dans le fichier de configuration de SeedFu (« db/seeds.rb »), nous lui demanderons d'exécuter le ou les fichiers d'insertion que nous voulons :

```ruby
User.destroy_all

SeedFu.seed("db/fixtures",/users/)

```

Figure 7 Exemple de fichier de configuration SeedFu

Dans l'exemple, je vide d'abord la table « User » avant d'insérer les données de base et d'ensuite insérer les données du fichier « users » qui se trouve dans le dossier « db/fixtures ».

Une fois que nous avons fini de configurer le fichier '*seeds.rb'*. Nous exécuterons la commande pour insérer nos données dans les tables :

```
$ rake db:seed
```

Figure 8 Commande d'insertion des données seed

6.3.3 Les relations/associations

Notre base de données et les modèles sont générés. Cependant il n'y a pas de relation entre les modèles.

Relations/associations

Une relation/association est un lien entre 2 tables, ce lien permet de créer des références, c'est-à-dire, par exemple, référencé un professeur à un topic et facilite l'utilisation des modèles. Il est possible de créer 2 types de relations/associations en RoR :

- One To One
- One To Many

Cependant, dans le cadre de l'application, nous avons utilisé que des associations one-to-many, donc je n'expliquerai que l'association « One-To-Many ».

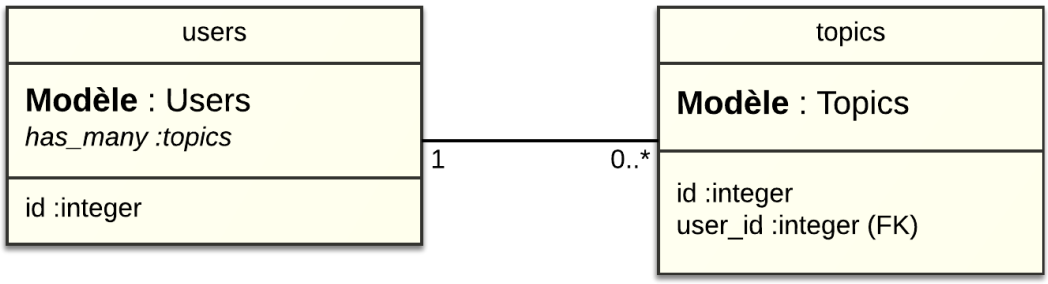
Créer une relation/association

Nous allons, par exemple, créer la relation entre le modèle « User » et « Topic ». Pour créer une relation, il faut utiliser les annotations :

- has_many
- belongs_to
- (et d'autres selon l'association que l'on désire)

Association One-To-Many : has_many / belongs_to

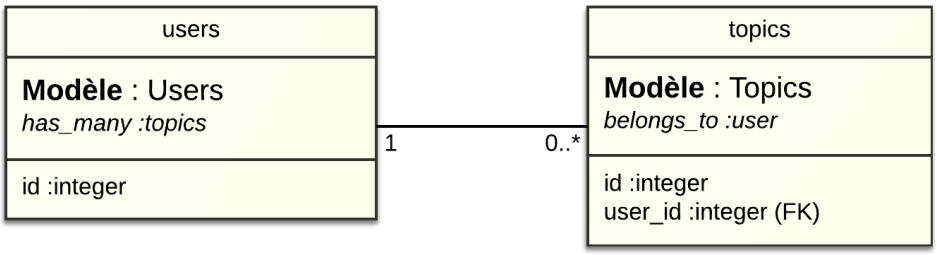
Pour créer une association one-to-many, il faut ajouter une annotation « has_many » au modèle qui possède la multiplicité.




```
class Users < ActiveRecord::Base
      has_many :topics
end
```

Cependant dans l'exemple, nous avons créé une association unidirectionnelle. Mais cette association doit être bidirectionnelle.

Pour que l'association soit bidirectionnelle, nous devons ajouter l'annotation « belongs_to » afin que depuis le topic, nous pouvons récupérer l'utilisateur.




```
class Users < ActiveRecord::Base
      has_many :topics
end

class Topics < ActiveRecord::Base
     belongs_to :user
end
```

Utilisation de ses associations

L'avantage de Rails n'est pas seulement d'ajouter les annotations pour créer les relations mais nous pouvons les utiliser dans le code.

Selon l'exemple, nous pouvons récupérer tous les topics qui possèdent la référence de l'utilisateur :

```
user = User.find(1)
user.topics
```

Comme il s'agit d'une relation bidirectionnelle, il est aussi possible de récupérer, depuis un topic, le créateur.

6.3.4 Routing

Qu'est-ce que le Routing ?

En RoR, le routing est le fait d'attribuer une requête venant d'un browser (par URL) à une action d'un contrôleur.

Fonctionnement

Quand votre application Rails reçoit une requête :

GET /topics/1

Le routeur de l'application va capturer le chemin et va identifier à quelle action de quel contrôleur l'a-t-on assignée. Ici, en l'occurrence, nous l'avons désigné à l'action « show » du contrôleur « topics » :

get '/topics/ :id', 'topics#show'

Ceci est une assignation classique et manuelle.

Création de routes

En RoR, nous avons une série d'action et de vue déjà défini lors de la génération des contrôleurs et des vues. Nous avons comme actions générées par défaut :

Nom de l'action
Index
Show
New
Edit
Create
Update
Delete

Mais elles ne possèdent pas encore de routes. Afin de créer ou assigner des routes à nos actions, nous devons configurer un fichier afin que notre application Rails sache quelle route est attribuée à quelle action et quel contrôleur. Nous devons configurer nos routes dans un fichier auto-généré à la création du projet : « config/routes.rb »

Voici un exemple de fichier routes.rb :

```ruby
Rails.application.routes.draw do

  devise_scope :user do
  end

  devise_for :users, :controllers => { :omniauth_callbacks => "users/omniauth_callbacks", sessions: "users/sessions", registrations:

  resources :labels

  resources :topic_joined_users

  resources :vote_user_comments

  resources :favor_surveys

  resources :labels_topics

  resources :user_answers

  resources :answers

  resources :comments

  resources :surveys

  resources :topics

  resources :users

  root to: "users#redirect_home"
  post '/language' => "users#change_language", as: "change_language"

  # User
  get '/home' => 'users#home' , as: 'home'
  get '/search' => 'users#search', as: 'search'

  # Topic
  post 'topics/create' => 'topics#create', as: "new_create_topic"
  post 'topics/create/closed' => 'topics#create_closed', as: "create_topic_closed"
  post 'topics/:id/close' => 'topics#close', as: "close_topic"
  post 'topics/:id/join' => 'topics#join', as: "join_topic"
```

Il existe différentes annotations afin de définir une route. Nous allons nous attarder seulement sur celle-ci :

- resources
- get / post / patch / put / delete
- root

L'annotation « resources » permet d'assigner des routes aux actions par défaut des contrôleurs, c'est-à-dire, assigner une route aux actions suivantes :

Routes	Nom de l'action	Type de requête
/contrôleur	Index	GET
/contrôleur/ :id	Show	GET
/contrôleur/new	New	GET
/contrôleur/ :id/edit	Edit	GET
/contrôleur	Create	POST
/contrôleur/ :id	Update	PATCH – PUT

/contrôleur/ :id	Delete	DELETE

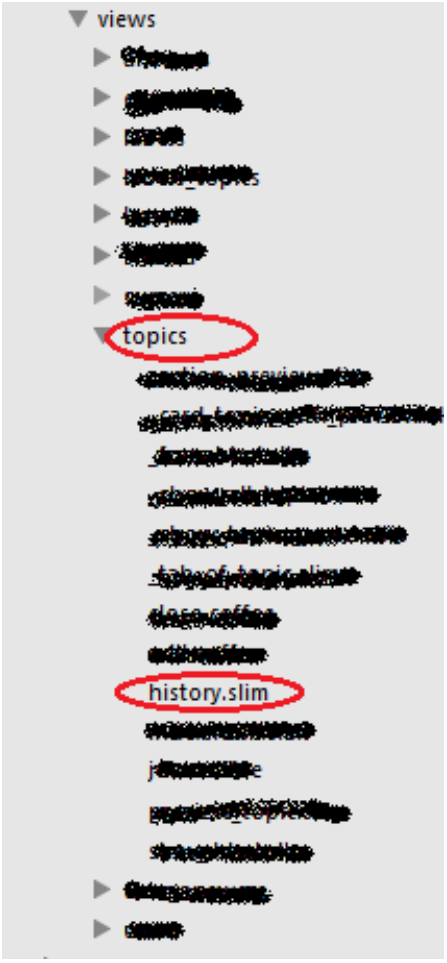
Nous avons maintenant nos actions par défaut qui ont tous une route d'accès. Maintenant nous désirons ajouter une action à notre contrôleur « Users », par exemple.

C'est ici qu'entre en jeu les annotations « get / post / patch / put / delete».
Prenons comme scenario suivant :

Nous souhaitons créer une page où un utilisateur peut retrouver tous les topics qu'il a joint, comme un historique de topics. Tout d'abord, nous allons créer l'action « history » dans le contrôleur « Topic » et effectuer le traitement nécessaire afin de récupérer ses topics qu'il a joint de cette année.

```
15  def history
16    @topics = TopicJoinedUser.where("user_id = ? AND strftime('%Y', created_at) >= ?", current_user.id, Time.now.year).paginate(:pag
17  end
```

Puisque nous avons créé une action et que la convention de RoR dis que pour chaque action il doit avoir une vue. Nous allons créer notre vue dans le dossier « topics » avec le même nom que notre action, afin que notre application puisse trouver la vue liée à l'action.



Voilà, maintenant nous avons notre action et notre vue. Dorénavant, nous devons donner accès à ceci pour l'utilisateur en assignant une route à notre action. Nous allons assigner notre route à une requête de type « get » car nous avons juste besoin de récupérer des informations.

Pour se faire, nous allons ajouter la ligne suivante à notre fichier config/routes.rb :

get ‘history’ => ‘topics#history’, as: ‘history’

L’argument ‘as: *** ‘ sert à attribuer un nom à notre route afin qu’on puisse l’utiliser pour un bouton ou autre.

À présent, nous avons notre action, notre vue associée à l’action et une route d’accès. L’utilisateur peut maintenant accéder à son historique de topics en allant sur l’url suivant :

http://example.com/history

L’annotation ‘root’ permet de définir une action à la route ‘/’ de l’application. Si un utilisateur accède à l’application sans forcément donner de route, il accédera à l’action défini par l’annotation ‘root’.

http://example.com/

6.3.5 Implémentation des interfaces

Pour l'implémentation de l'interface et des interactions, nous avons utilisé un autre langage que de l'HTML, CSS et JavaScript. Nous avons utilisé :

- Slim au lieu de l'HTML
- SASS au lieu du CSS
- CoffeeScript au lieu de Javascript

Slim

Slim est un langage utilisé pour développer des vues comme de l'HTML. Par contre, le slim génère de l'HTML. En effet, Slim est, si l'on veut, de l'HTML condensé mais il va être compilé en HTML. Il fait abstraction de tous ce qui est logique, c'est-à-dire, les signes « < » « > », les balises fermantes, etc… Il se base sur l'indentation pour la structure du code. Exemple de code Slim :

a#id_btn.btn href='/history'

Donc là nous avons créé une ancre avec comme class 'btn' et comme href '/history'. Pour l'assignation d'une classe et d'un id, on peut utiliser les accesseurs css, c'est –à-dire assigner une classe à l'aide d'un point ou un id à l'aide d'un #. Et ce code slim génère le code html suivant :

****

SASS

Même idée que Slim, SASS est un langage utilisé pour tous ce qui est code « stylesheet » à la place de CSS. SASS fait abstraction de symboles ou de choses logiques. Voici un exemple de code SASS :

SASS	CSS
.logo **background: transparent**	**.logo {** **background : transparent;** **}**

SASS permet aussi de créer des ensembles d'attribut, variable ou des 'méthodes'. En effet, nous pouvons créer des 'méthodes'. Dans le jargon SASS, on appelle cela des 'mixin'.

```
35  @mixin max-size($max-height, $max-width)
36    max-height: $max-height !important
37    max-width: $max-width !important
```

Ici nous avons créé un 'mixin' qui permet d'assigner une largeur maximale et une hauteur maximale à l'objet dans lequel on appelle cette 'mixin'.

```
132  .img-profil-mobile
133    @include max-size(100px, 100px)
```

Ainsi nous assignons à tous les objets ayant la classe « img-profil-mobile » une taille maximale de 100x100.

En SASS, il est possible aussi de créer des variables et les utiliser à la valeur d'un attribut.

```
$font-stack:    Helvetica, sans-serif
$primary-color: #333

body
  font: 100% $font-stack
  color: $primary-color
```

CoffeeScript

Coffeescript est un petit langage, comme Slim et SASS, qui permet d'écrire du code plus simplement et qui se compile en JavaScript. L'idée est toujours la même que Slim et SASS. Le langage fait abstraction de symboles et autres afin d'alléger le code tout en gardant une bonne lisibilité.

CoffeeScript	Javascript
`# Functions:` `square = (x) -> x * x`	`square = function(x) { return x * x; };`

Utilité

Ainsi grâce à cette petite panoplie de langage, nous pouvons générer des vues très rapidement tout en conservant du code lisible et compréhensible. Cependant, comme ces langages font abstraction de certaine chose, le développeur devra faire attention à bien indenter ou écrire le code puisque l'écriture est simple, les erreurs peuvent vite tomber.

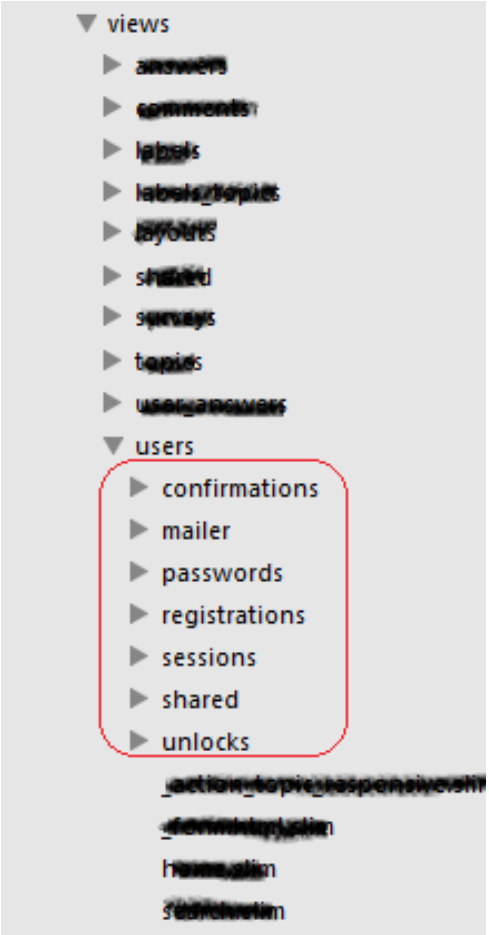
6.3.6 Devise

Devise [5]est une gem très connue dans le monde de Ruby On Rails. Elle permet d'ajouter un système complet d'inscription/authentification à une application Rails.

Installation

Premièrement, il fonctionne avec la base de données, logique. Il vous faudra créer une table, ou l'associer à la table 'User', qui peut stocker les informations des utilisateurs -qui doivent se connecter- comme le pseudo, l'email, le mot de passe, etc…

Deuxièmement, il génère une série de fichiers définis pour les interfaces de connexion, inscription, oubli de mot de passe, validation de compte, etc… Puisque dans la logique, nous assignons les informations de devise à la table « User », alors Devise génère les vues dans le dossier 'view/users'



Dernièrement, il faut configurer les contrôleurs. En effet, maintenant il s'agit d'assigner à quel action/page nous souhaitons que le client soit connecté à l'aide de l'annotation suivante :

before_action :authenticate_user !

[5] https://github.com/plataformatec/devise

Une fois le système mise en place, Devise fonctionne avec la session.

Lorsqu'un utilisateur se connecte via le système de DEVISE. Il va récupérer toutes les informations nécessaires à la gestion d'une authentification, au navigateur du client, tel que la session, cookies, etc…

Dès que les informations nécessaires ont été prises. DEVISE va ensuite créer des variables global à l'application, des variables d'environnements et des méthodes utilitaires pour le développement.

- current_user : Renvoie l'utilisateur connecté
- user_signed_in? : méthode qui renvoie 'true' si un utilisateur est connecté
- user_session : variable qui donne accès aux informations de la session de l'utilisateur.

6.3.7 Tests fonctionnels

Un test fonctionnel est un type de test où on vérifie si le chemin de navigation est correct ou si toutes les fonctionnalités d'une page sont biens en place et opérationnelles.

Pour les tests, nous utilisons RSPEC. Pourquoi ne pas utiliser le système de test de base qu'offre Rails. Tous simplement, parce que mon promoteur m'a proposé RSPEC et que je l'ai trouvé sympa et simple à utiliser.

RSPEC

RSPEC est une gem qui permet de créer des tests et qu'on définit sous forme de 'scenario'.

En effet, grâce à cette gem, nous pouvons définir un ensemble de scenario afin de simuler au maximum l'utilisation d'un utilisateur dans un navigateur. Afin de créer des tests fonctionnels ou d'interaction, nous avons besoin de la gem 'Capybara'.

Capybara

Capybara est une gem qui permet de créer des tests fonctionnels ou d'interactions en simulant comme si un utilisateur était en train d'utiliser notre application. Il utilise des drivers afin de pouvoir tester au mieux l'application. Dans notre cas, nous utilisons le driver poltergeist et le navigateur PhantomJS.

Qu'est-ce que poltergeist et PhantomJS ?

Poltergeist est un driver utilitaire pour l'environnement test. Elle permet de simuler un utilisateur sur un navigateur mais de manière légère sans qu'il crée une espèce de 'robot' qui pourrait être lourd pour la machine. Il utilise PhantomJS comme navigateur. Nous pouvons aussi à l'aide de cette gem, pouvoir générer des captures d'écran et/ou récupérer le code HTML lors d'un échec d'un test.

PhantomJS est un navigateur internet sans interface utilisateur. Il est essentiellement utilisé dans les environnements test. Il est souvent utilisé par des drivers tests. Il est appelé 'navigateur fantôme'.

Comme il est considéré comme un navigateur, il exécute le Javascript, CSS et HTML de votre page même en environnement test malgré qu'il n'ait pas d'interface.

Ainsi grâce à ces 2 gems, nous pouvons aller plus loin dans les tests. Nous pouvons tester les fonctionnalités de notre application mais aussi en même temps l'interface : vérifier si le JavaScript s'exécute bien, si un élément est au bon endroit, etc…

Fonctionnement

Tout d'abord, il faut régler dans les fichiers de configurations de RSPEC, qu'on spécifie que le driver de Capybara soit bien poltergeist et PhantomJS comme navigateur fantôme.

```
47    Capybara.default_driver = :poltergeist
48    Capybara.javascript_driver = :poltergeist
49    Capybara.register_driver :poltergeist do |app|
50      Capybara::Poltergeist::Driver.new(app, :phantomjs => Settings.phantomjs.path, :js_errors => false, :timeout => 180)
51    end
```

Une fois RSPEC configuré, nous pouvons nous attacher au test. Il faut créer un fichier test par contrôleur et tester un contrôleur à la fois.

Ensuite, il est possible de créer ce qu'on appelle des 'scenario'. Ces scenarios sont, ici, des simulations d'interaction de l'utilisateur sur le navigateur. Nous pouvons assigner des noms à nos scénarios mais cela permet juste de savoir quel test a échoué s'il y a eu un échec dans un des tests.

```ruby
require 'rails_helper'
require 'spec_helper'

RSpec.describe UsersController, type: :controller do

    before do
      Rails.application.env_config["devise.mapping"] = Devise.mappings[:user] # If using Devise
    end

    describe do
        scenario "log-in - teacher" do
            log_in_as_teacher
            assert page.find("#log-out")
        end

        scenario "check home - teacher" do
            log_in_as_teacher
            assert page.find('.add-new-topic')
        end

        scenario " check home - student " do
            log_in_as_student
            expect(page).not_to have_css('.add-new-topic')
        end
    end
end
```

Voici un exemple de fichier test pour le contrôleur 'User'.

Nous pouvons créer plusieurs scenarios pour un même contrôleur, autant de scenario que l'on veut.

Le bloc « before do » est un bloc de code qui va être exécuté avant tous les tests.

7 Conclusion

Ces 15 semaines à l'ECAM ont été une expérience incroyable et très enrichissante. Elles m'ont permis de m'enrichir dans l'univers du développement WEB. J'ai aussi appris beaucoup de langages et Frameworks utiles tels que Rails, Ruby, CoffeeScript, Slim, etc…

Cela m'a aussi donné une expérience du monde du travail. J'ai découvert et appris le rythme de vie d'un employé qui, selon moi, est très différent du statut d'étudiant car en tant qu'étudiant, il y a des choses qui sont + ou – permis, tandis qu'en tant qu'employé, ces choses sont littéralement interdit ou même très déconseillé. J'ai aussi appris que les deadlines étaient très importants dans le développement d'une application.

En ce qui concerne l'application, il s'agit d'une application web qui permettait d'aider un professeur, à un cours, pour gérer les commentaires/questions des étudiants ou/et de questionner ceux-ci. Elle devait être simple d'utilisation, mis à jour en temps réel, rapide et clair. L'application avait énormément de point à avoir. Malheureusement, en 15 semaines, tous les points n'ont pu être atteins.

Atteinte des objectifs

Comme énoncé ci-dessus, tous les points n'ont pu être atteints pour diverse raison. L'implémentation de ceux-ci était soit lourde, longue et nécessitait une certaine connaissance très accrus dans le domaine ou soit tout simplement par manque de consigne précise.

Nous avons donc dû faire des choix, nous avons dû sélectionner seulement les demandes qui ont été claires, non ambigües et importantes. Cependant, avec déception, je n'ai pas pu optimiser toutes les fonctionnalités que nous avons choisies par soucis de temps.

Futur de l'application

Comme l'application n'est pas finie par manque de fonctionnalités ou fonctionnalités non-optimisées. L'informaticien, qui était en charge de moi, va récupérer l'application. Il va finir les fonctionnalités manquantes et non-optimisées, les peaufiner et achever les différentes versions. Je ne suis au courant que de quelques fonctionnalités manquantes tel que : les statistiques, la gestion des mails, un système de partage de document dans un topic, un système de présence et plein d'autre que je ne me souviens plus.

Avis de l'application

L'idée de l'application m'a plu dès le départ. Mais au vue des nombreuses fonctionnalités demandées par le client et de toutes les technologies que j'allais devoir apprendre, j'étais au courant dès le départ que je ne pourrais pas terminer la totalité de l'application.

Cependant, malgré cela, j'ai quand même appris des technologies très intéressantes comme Ruby On Rails, Faye, TMUX, Vim, Sublime Text et des langages agréables et simples à utiliser tels que Ruby, CoffeeScript, Slim, YAML,…

Le choix de ces technologies m'ont été impressionnantes car j'ai pu découvrir qu'en les combinant avec les langages imposés, nous pouvions déployer une application web en très peu de temps. Malheureusement, je ne suis pas un fan du développement sans souris qu'offre VIM. Alors en cours de route, j'ai eu l'autorisation de passer à Sublime Text tout en restant dans l'environnement Linux. Malgré que j'estime, maintenant, que VIM est un éditeur incroyable en ajoutant les plugins nécessaires et que je comprends pourquoi il est toujours opérationnel depuis plus de 20 ans.

8 Glossaire

- **Rails** : Framework web libre écrit en Ruby. Il permet de créer des applications web et de les déployer assez rapidement.

- **Topic** : Un topic est un espace dédié à un cours où un utilisateur peut y mettre des commentaires et/ou un professeur peut soumettre des questions ou répondre aux commentaires.

- **Sondage** : Un sondage est une question libre posée à un ensemble de personne dont les réponses sont souvent définies par celui qui a créé la question.

- **RSPEC** : Gem permettant de créer des tests.

- **DEVISE** : Gem très importante dans l'univers de Ruby On Rails. Elle permet d'insérer un système d'authentification à l'application.

- **Slim** : langage de type template utilisé pour écrire des vues à la place de l'HTML. Mais lorsque ce langage est compilé, le code est converti en HTML. Nous pouvons dire que c'est de l'HTML condensé.

- **CoffeeScript** : Petit langage qui compile en JavaScript. Nous pouvons dire que c'est du JavaScript condensé.

- **SASS** : langage utilisé de façon similaire à du CSS mais en format condensé.

9 Bibliographie

- Livre

https://pragprog.com/book/dnvim/practical-vim

- Internet

http://guides.rubyonrails.org/

http://apidock.com/rails

http://apidock.com/ruby

http://materializecss.com/ : Un peu comme Bootstrap dans le système de balise CSS, sauf qu'ils utilisent la technologie de Google, Material Design.

http://railscasts.com/ : Site qui aide les développeurs en RoR. Ce site montre en vidéo ou en pdf comment installer, utiliser ou configurer différentes fonctionnalités de RoR.

http://railsforzombies.org/ : Site interactif permet d'apprendre Rails, les requêtes, l'architecture sous forme de mini-jeu et mini-projet en ligne.

http://tryruby.org/levels/1/challenges/0 : Site sous forme de jeu qui permet d'apprendre la syntaxe, les conventions, etc… de Ruby.

http://docs.railsbridge.org/intro-to-rails/setting_the_default_page

http://sass-lang.com/guide

http://faye.jcoglan.com/

http://coffeescript.org/

http://slim-lang.com/

https://github.com/plataformatec/devise

https://github.com/rspec/rspec-rails

https://github.com/jnicklas/capybara

https://github.com/teampoltergeist/poltergeist

http://phantomjs.org/

http://code.macournoyer.com/thin/