Ville de Liège
École de Commerce et d'Informatique
Enseignement de promotion sociale
Rue Hazinelle 2 - 4000 LIÈGE
Matricule : 6.188.038
04/221 37 86

Réalisation d'une application de suivi des contacts externes du Conseil supérieur de l'audiovisuel

Travail de fin d'études présenté par
Laurent RICHARD
En vue de l'obtention du diplôme de
Gradué en Informatique

Année académique 2007-2008

Life is a succession of lessons,
which must be lived to be understood.
— *Ralph Waldo Emerson*

Remerciements

Hélène Thomas a dit : « *Le monde devrait remercier cette étonnante cohorte de gens qui font toujours preuve d'une insolente et illogique gentillesse.* »

N'étant pas le monde mais ayant reçu une aide et des conseils avisés de certaines personnes, il me semblerait inadéquat de ne pas les remercier pour leur apport.

S'il est bien une personne que je dois remercier, c'est ma future épouse Nathalie. Tant d'années à supporter mes absences pour me permettre d'obtenir ce graduat.

Je tiens également à remercier Monsieur Jacques Thoorens qui a accepté d'encadrer ce travail de fin d'études. Par son ouverture d'esprit, la confiance qu'il m'a accordée, je me suis senti libre de réaliser ce travail tel que je l'envisageais.

Je tiens aussi à remercier Monsieur Bernard Dubuisson, ma personne ressource auprès du client, qui m'a donné l'opportunité de réaliser le travail de fin d'études au Conseil Supérieur de l'Audiovisuel. Notre collaboration fut efficace malgré nos emplois du temps.

Sans pour autant les oublier, je tiens pour finir à remercier Axel Krausch-Lacroix pour sa relecture et ses commentaires ainsi que Rémi Laurent pour ses conseils techniques en Ruby on Rails.

Table des matières

Chapitre 1

Introduction

Oh, this is gonna be fun ! We can stay up late, swaping manly stories, and in the morning, I'm making waffles !
— *Donkey, Shrek*

1.1 Motivations et problématique

Ce projet de collaboration avec le CSA est né d'un intérêt commun pour le langage Ruby[1] et le framework Ruby on Rails[2].

Ruby on Rails est un framework libre dédié au développement d'applications Web/Ajax, au sens large du terme. Même s'il a encore tout à prouver sur le marché, il suscite un grand intérêt pour le client en termes de réutilisation du code et d'accessibilité de celui-ci pour une personne qui peut être formée assez facilement . Ce qui augure des gains de productivité et une réduction des délais de développement.

Le langage Ruby date pourtant de la même époque que Java (milieu des années 90).

Les avantages de Ruby pour le client sont également :

- un code très compact ;
- une syntaxe proche du langage naturel ;
- un métalangage permettant d'écrire aisément des langages métiers.

Le framework Ruby on Rails est déjà utilisé pour le site Internet du client. Tous les futurs développements seront également réalisés avec celui-ci.

Avec Ruby on Rails, un seul et même langage suffit à tout faire car il génère tous les codes nécessaires. Les conventions sont en outre préférées aux configurations. Le mapping objet relationnel s'en trouve très simplifié.

Parmi les autres atouts, l'architecture MVC (Modèle Vue Contrôleur) est systématiquement mise en oeuvre et des squelettes de tests unitaires et fonctionnels sont générés.

Quand une personne ayant un grand intérêt pour ce langage qui a le vent en poupe et qui attire de plus en plus de développeurs rencontre un organisme souhaitant créer une application sous un framework dérivé de ce langage se rencontrent ... cela aboutit à un projet passionnant.

1.2 Organisation du travail de fin d'étude

Tout a commencé via des contacts téléphoniques, des messages électroniques ainsi que des visites sur place pour mieux cerner les attentes du client en termes précis avec une analyse fonctionnelle par classe.

[1] http ://www.ruby-lang.org
[2] http ://www.rubyonrails.org/

Au fur et à mesure de la conception, les questions relatives à la fonctionnalité, l'ergonomie et les données essentielles, ... se sont affinées.

Afin de pouvoir mettre le projet en production à tout moment, nous avons décidé de réaliser le projet par itération. Cela a nécessité la révision des attentes de départ pour obtenir un programme minimal qui serait amélioré au fur et à mesure par ajout de fonctionnalités.

A chaque moment, le client était partie prenante des réflexions afin d'aboutir à un produit fonctionnel et performant en termes de maintenabilité, de sécurité et de modularité.

Il ne fut pas possible de tout mettre en place (historique des occupations des personnes, fichiers Excel complets) mais chacune de ces fonctionnalités peut être intégrée sans trop de soucis au programme existant.

Le client s'est estimé très satisfait du travail accompli et de la bonne communication entre les parties.

1.3 Organisation du manuscrit

Ce document écrit représente de manière fidèle le cheminement du projet, en partant de l'analyse préalable et fonctionnelle réalisée avec le client, jusqu'aux tests unitaires et fonctionnels permettant de mieux maintenir le programme à l'avenir. Certaines pistes sont encore à envisager.

J'ai fait en sorte de ne rien laisser au hasard. Chaque point que j'ai jugé essentiel pour le travail sera abordé dans ce document à savoir :

- le feedback des utilisateurs ;
- l'importance des tests (création et modification) tout au long des itérations ;
- l'importance des principes de sécurité.

1.4 Description détaillée de l'organisme

Le Conseil supérieur de l'audiovisuel[3] (CSA) est chargé de la régulation du secteur de la radiodiffusion en Communauté française de Belgique. Ses missions sont principalement de contrôler le respect des obligations :

- des éditeurs de services (RTBF, télévisions locales, télévisions et radios privées) ;
- des distributeurs de services (câblodistributeurs, Belgacom, Be TV, Proximus, Mobistar, ...) ;
- des opérateurs de réseaux (câblodistributeurs, Belgacom, RTBF, ...).

Le CSA est composé de deux Collèges : une instance d'avis (Collège d'avis) chargée de rendre des avis sur toute question relative à l'audiovisuel et une instance décisionnelle (Collège d'autorisation et de contrôle) chargée :

- d'attribuer les autorisations d'émettre aux télévisions et radios privées établies en Communauté française ;
- de contrôler le respect des obligations des éditeurs de services, des distributeurs de services et des opérateurs de réseaux ;
- de sanctionner les infractions à ces obligations.

1.4.1 Le bureau

Le Bureau du CSA est composé de la présidente, des trois vice-présidents et du directeur. Ce dernier est chargé des décisions opérationnelles. Les membres du Bureau sont également membres de droit du Collège d'autorisation et de contrôle et du Collège d'avis.

[3] http ://www.csa.be

1.4.1.1 Mission

Le Bureau a le pouvoir d'accomplir, de façon autonome, tous les actes nécessaires ou utiles à l'exercice des compétences du CSA et à son administration. Il le représente en justice et à l'égard des tiers, peut contracter en son nom et en recrute le personnel, auquel il délègue certaines de ses attributions (gestion, préparation des travaux des Collèges, exécution des décisions, ...).

Le Bureau coordonne et organise les travaux du CSA, veille à la conformité des avis au droit interne et européen ou international et résout les conflits de toute nature qui apparaissent entre les Collèges.

Pour accomplir ces missions, le Bureau peut faire au Gouvernement toutes les recommandations qu'il juge utiles. Il peut aussi faire appel à des services extérieurs ou à des experts en vue d'aider le CSA et les Collèges dans l'exercice de leurs missions.

Le président préside de droit tous les Collèges et les vice-présidents assistent avec voix délibérative à toutes les réunions des Collèges.

1.4.1.2 Composition

Le Bureau est composé du président et de trois vice-présidents, désignés par le Gouvernement. Leur mandat est d'une durée de cinq ans, renouvelable.

Les membres du Bureau sont soumis aux mêmes règles d'incompatibilités que les membres du Collège d'autorisation et de contrôle et le personnel du CSA.

La composition du Bureau garantit la représentation des différentes tendances idéologiques et philosophiques.

1.4.2 Le collège d'avis

1.4.2.1 Mission

Le Collège d'avis a pour mission principale de rendre, d'initiative ou à la demande du Gouvernement ou du Parlement de la Communauté française, des avis sur toute question relative à l'audiovisuel, en ce compris la communication publicitaire (à l'exception des questions relevant de la compétence du Collège d'autorisation et de contrôle).

Il est en outre chargé de se prononcer sur :

1. les modifications décrétales et réglementaires que semble réclamer l'évolution technologique, économique, sociale, culturelle des activités du secteur de l'audiovisuel, ainsi que du droit européen et international ;
2. le respect des règles démocratiques garanties par la Constitution ;
3. la protection de l'enfance et de l'adolescence dans la programmation des émissions.

Il doit également rédiger et tenir à jour des règlements portant sur la communication publicitaire, sur le respect de la dignité humaine, sur la protection des mineurs et sur l'information politique en périodes électorales.

1.4.2.2 Composition

Outre les membres du Bureau (le président et les trois vice-présidents), le Collège d'avis est composé de 30 membres (ayant chacun un suppléant) désignés par le Gouvernement. Leur mandat est d'une durée de quatre ans, renouvelable.

Ces membres et leurs suppléants sont des professionnels issus des différents secteurs de l'audiovisuel (éditeurs et distributeurs de services de radio et de télévision, opérateurs de réseaux, cinéma, sociétés d'auteurs, producteurs, régies publicitaires, annonceurs, associations de consommateurs, sociétés de presse, journalistes, ...).

La composition du Collège garantit la représentation des différentes tendances idéologiques et philosophiques.

Assistent aux travaux avec voix consultative deux délégués du Gouvernement, le Secrétaire général du Ministère de la Communauté française ou son représentant, trois délégués du Conseil d'éducation aux médias, ainsi que les présidents et vice-présidents sortants.

1.4.3 Collège d'autorisation et de contrôle

1.4.3.1 Mission

Le Collège d'autorisation et de contrôle exerce deux types de compétences : l'une d'autorisation, l'autre de contrôle. Ce dernier pouvoir est assorti de celui de sanctionner l'éditeur de services, le distributeur de services ou l'opérateur de réseau en cas de manquement à ses obligations légales ou conventionnelles.

Il est donc chargé de :

- autoriser les éditeurs de services - sauf la RTBF et les télévisions locales - et l'usage de radiofréquences ;
- rendre un avis préalable à l'autorisation par le Gouvernement de la Communauté française de Belgique de télévisions locales, mais aussi et surtout de rendre un projet de convention à conclure entre le Gouvernement et un éditeur de service ;
- rendre, au moins une fois par an, un avis sur la réalisation des obligations découlant du contrat de gestion de la RTBF et des obligations des télévisions locales, ainsi que des obligations découlant des conventions conclues entre Gouvernement et éditeurs de services bénéficiant d'un droit de distribution obligatoire ;
- faire des recommandations de portée générale ou particulière ;
- constater toute infraction aux lois, décrets et règlements en matière d'audiovisuel et toute violation d'obligation conventionnelle ;
- déterminer les marchés pertinents et les opérateurs de réseau puissants sur le marché ainsi que leurs obligations ;
- en cas d'infraction, prononcer une sanction administrative allant de l'avertissement au retrait de l'autorisation.

1.4.3.2 Composition

Outre les membres du Bureau (le président et les trois vice-présidents), le Collège d'autorisation et de contrôle est composé de six membres, dont trois sont désignés par le Conseil de la Communauté française et trois par le Gouvernement. Leur mandat est d'une durée de quatre ans, renouvelable.

Ces membres sont choisis parmi des personnes reconnues pour leurs compétences dans les domaines du droit, de l'audiovisuel ou de la communication, mais qui ne peuvent y exercer une fonction de nature à créer un conflit d'intérêt personnel ou fonctionnel.

La composition du Collège garantit la représentation des différentes tendances idéologiques et philosophiques.

Le Secrétaire général du Ministère de la Communauté française assiste aux travaux du Collège avec voix consultative.

1.4.4 Services du CSA

Le personnel du CSA est chargé de préparer les dossiers soumis à la décision des Collèges, sous la responsabilité du directeur. Les incompatibilités prévues à l'article 136 du décret du 27 février 2003 sur la radiodiffusion sont d'application pour tous les membres du personnel.

1.4.4.1 Mission

Le personnel du CSA assure la préparation des travaux et l'exécution des décisions du Bureau et des Collèges.

Sur délégation du Bureau, son Directeur assure également la gestion quotidienne.

Au sein du personnel, un service spécifique, le Secrétariat d'instruction, reçoit les plaintes ou les remarques du public concernant les programmes de radio ou de télévision :

- atteintes à la dignité humaine ;
- violence gratuite ;
- protection des mineurs ;
- application de la signalétique ;
- durée de la publicité ;
- ...

Il instruit toutes les plaintes qui lui sont adressées puis les soumet au Collège d'autorisation et de contrôle, qui peut constater l'infraction et, le cas échéant, la sanctionner. En vue d'assurer les missions qui lui sont confiées, le secrétariat d'instruction peut recueillir tant auprès de personnes physiques que de personnes morales toutes les informations nécessaires pour s'assurer du respect des obligations imposées aux titulaires d'autorisation ; il peut également procéder à des enquêtes.

1.4.4.2 Composition

Le cadre du personnel du CSA a été fixé par le gouvernement à 28 agents. 21 agents sont aujourd'hui affectés au CSA.

Les membres du personnel sont soumis aux mêmes règles d'incompatibilités que les membres du Collège d'autorisation et de contrôle.

1.5 Description du fonctionnement actuel du domaine

Le CSA est un organisme indépendant d'État. Par cela, il n'a aucune concurrence dans son domaine.

FIG. 1.1 – Source : http ://www.csa.be

1.6 Présentation des problèmes actuels à améliorer ou résoudre.

1.6.1 Situation de départ

Le CSA est amené à communiquer, sous diverses formes, avec un grand nombre de contacts (plusieurs centaines).

Les différents types d'intervenants sont :

- les membres des collèges du CSA. Il s'agit de personnes qui ne sont pas membres du personnel mais qui ont des contacts et réunions réguliers au sein de l'institution ;
- des citoyens, des particuliers (plaintes, questions) ;
- des organismes du secteur de l'audiovisuel : éditeurs de services, distributeurs de services, opérateurs de réseaux ;
- au sein de ces organismes, des personnes de contacts diverses ;
- des organisations : fédérations sectorielles, groupes d'intérêts, ONG, etc. ;
- d'autres institutions : régulateurs, gouvernements, parlements, administrations belges et étrangers, nationaux et supranationaux ;
- des fournisseurs divers.

Les différentes formes de communication sont :

- le téléphone ;
- le courrier électronique ;
- le courrier postal, individuel ou par publipostage ;
- le contact direct (visite sur place ou chez le contact).

1.6.2 Problèmes du système actuel

Actuellement, l'ensemble de ces contacts est répertorié par le bais de plusieurs fichiers Microsoft Excel. Ceux-ci ne sont pas mis en réseau via le partage Windows. Le parc informatique est également hétéroclite (Microsoft Windows, Mac OS, Linux).

Ce mode de fonctionnement comporte de nombreux désavantages, en particulier des problèmes de performances, de cohérence, d'intégrité de données, d'interopérabilité.

L'objectif de l'application est de fournir une solution centralisée qui permettra à la fois d'exploiter les données de contacts à des fins diverses, mais aussi de conserver l'intégrité et la cohérence des données.

Sa mise en place dans un Intranet permettra à tous d'y accéder de manière sécurisée, selon leur autorisation, sur n'importe quel type de machine tout en garantissant une unicité au point de vue du lieu de stockage de l'information.

Chapitre 2

Spécification des besoins initiaux du client

- La solution souhaitée repose sur une base de données relationnelle qui reprendra l'ensemble des informations pertinentes pour la gestion des contacts. Cette base de données sera accessible par le biais d'une application de type client serveur développée au moyen de Ruby on Rails ;

- L'application gérera notamment les droit d'accès et les permissions des utilisateurs, la complexité des données (personnes et organismes, situations de double casquettes, catégories ouvertes) et leur intégrité (minimisation de la redondance des informations stockées dans la base de données) ;

- L'application gérera également des procédures de prévention des erreurs et/ou de tolérance à l'erreur à l'encodage. Par exemple, l'application avertira l'utilisateur lorsqu'il souhaite encoder un contact déjà présent dans la base de données ;

- L'application permettra également une exploitation optimale des données comme leur exportation à diverses fins, et en particulier :

  – envoi de courrier électronique ;
  – génération de modèles de courrier papier pré-adressés ;
  – génération de fichiers de base pour constituer un mailing de courrier papier ou par courrier électronique.

- L'application permettra de gérer l'ensemble des données liées à un contact particulier, qu'il s'agisse d'un organisme ou d'une personne physique. Elle permettra de catégoriser chaque contact selon divers critères ;

- Le projet impliquera également la récupération d'un maximum de données existantes, à l'heure actuelle sous forme de fichiers Microsoft Excel structurés.

Chapitre 3

Analyse préalable avec le client

3.1 Organismes

Un organisme est une personne morale, une société, une administration, une institution, une association (ASBL ou de fait, identifiée principalement par sa raison sociale).

Un organisme peut avoir plusieurs adresses : un siège social, ainsi qu'éventuellement X sièges d'exploitation, ou simplement une adresse de référence sans savoir quel statut a cette adresse. Un organisme doit avoir une adresse par défaut.

Un organisme peut avoir zéro, une ou plusieurs fonctions, qui sont autant de rôles exercés en son sein. Par exemple, une fonction de Président, de Directeur financier, d'Administrateur, de Commissaire, de Conseiller, etc. Il n'existe pas de liste standardisée des fonctions, même si certaines seront plus fréquentes que d'autres. Un organisme a une Fonction par défaut, s'il en existe au moins une.

Un organisme peut appartenir à zéro, une ou plusieurs catégories d'organismes.

3.2 Personnes

Le terme « Personne » désigne une personne physique. Elle a au moins un nom, et souvent un prénom s'il est connu. Une personne peut être renseignée à titre particulier et/ou rattachée à un ou plusieurs organismes via une fonction. Ces combinaisons peuvent varier avec le temps.

Une personne peut avoir une ou plusieurs fonctions, elles-mêmes rattachées ou non à un organisme. Par exemple « Marc Focroulle » est « Commissaire du Gouvernement ». La ou les fonctions d'une personne peuvent varier avec le temps. Par défaut, il est créé au moins une fonction pour chaque personne.

Une personne pourra avoir une ou plusieurs adresses selon ses différentes fonctions. Ces adresses pourront, selon le cas, être rattachées soit à la fonction (dans le cas où la personne a une fonction qui n'est pas rattachée à un organisme), soit à l'organisme (dans le cas où la personne a une fonction au sein d'un organisme).

Une personne peut appartenir à zéro, une ou plusieurs catégories.

3.3 Fonctions

Une fonction est une propriété des organismes et des personnes. Dans la plupart des cas, la fonction jouera le rôle de table d'association entre organismes et personnes. Toutefois, une fonction pourrait être associée uniquement à une personne (dans le cas d'un particulier ou de toute personne qui ne représente que lui-même), ou uniquement à un organisme (dans le cas où le titulaire de ce poste est inconnu).

Une fonction peut être rattachée :

- soit à un organisme (« NRJ Belgique SA » a un « Directeur ») ;
- soit à une personne (« Marc Focroulle » est « Commissaire du Gouvernement ») ;
- soit à un organisme et une personne (« Be TV SA » a un « Directeur technique » qui est « Francis Bodson »).

Une fonction peut changer de mains : la fonction « Directeur » de « NRJ Belgique SA » a été occupée par « Eric Adelbrecht » d'une date indéterminée jusqu'au 01/09/2006. A partir de cette date, cette fonction est inoccupée, la fonction de référence devient donc celle de « Directeur technique », « Mathieu Sibille ». A partir du 1-01-2007, la fonction de « Directeur » de « NRJ Belgique SA » est occupée par « Bruno Van Sieleghem ».

Cet historique doit être conservé, et l'on doit pouvoir retrouver :

- l'historique des personnes ayant occupé une Fonction ;
- l'historique des fonctions exercées par une Personne ;
- l'historique des fonctions créées et supprimées pour un Organisme ;

Le titre d'une fonction (par exemple « Directeur d'antenne ») est modifiable : on peut le corriger si un titre a été encodé de manière erronée (par exemple remplacer « Directeur d'antenne » par « Rédacteur en chef »). Par contre, si une personne qui était jusqu'à présent « Rédacteur en chef » est promu « Administrateur délégué », il s'agit d'une nouvelle fonction à créer, de sorte que l'ancienne fonction sera conservée dans l'historique de la personne et de l'organisme.

3.4 Adresses

Une adresse est une entité en soi qui peut être associée soit à un organisme, soit à une fonction. Il sera éventuellement nécessaire de prévoir des sous-objets, par exemple un nombre indéterminé de champs adresse e-mail.

L'association d'une adresse à un autre objet a une dimension temporelle. L'adresse peut changer au cours du temps. L'historique des anciennes adresses devra être conservé tout comme celui des fonctions.

3.5 Catégories

Les catégories sont un système de « tagging » permettant d'associer un mot-clé commun à plusieurs organismes, personnes ou fonctions, de manière à pouvoir les regrouper et les retrouver facilement.

Par exemple, l'organisme « NRJ Belgique SA » pourra être associé aux catégories « Radios », « Éditeurs de services », « Éditeurs de radiodiffusion sonore autre que FM », « Réseaux FM », « Participants Groupe de travail numérique », etc.

Il existera deux familles de catégories : celles associées aux organismes, et celles associées aux personnes et fonctions. Ces deux familles de catégories peuvent correspondre à deux réalités différentes, c'est pourquoi il y aura probablement peu de recoupements.

Les catégories seront ouvertes. Il sera possible d'en créer de nouvelles, mais un système de suggestion à l'encodage devra permettre de conserver une certaine cohérence. De cette manière, chaque utilisateur aura, s'il le souhaite, le loisir de recourir à ses propres catégorisations en fonction de son travail.

3.6 Utilisateurs

L'utilisateur est la personne qui, en tant que membre du personnel du CSA, dispose des permissions nécessaires pour utiliser le système. Un utilisateur correspond obligatoirement à une et une seule personne.

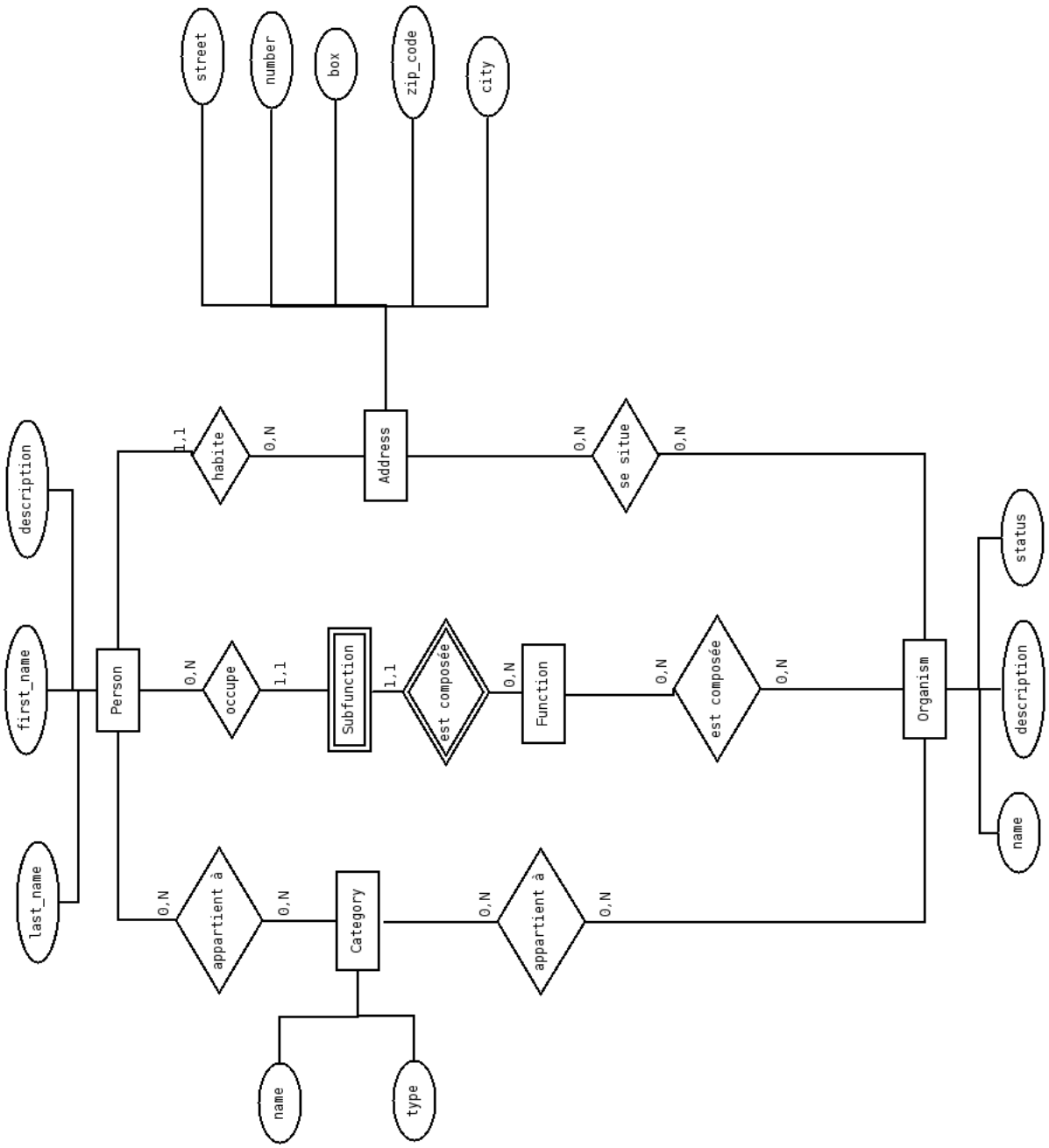
On peut distinguer plusieurs niveaux de permissions définies pour un utilisateur, ceci afin de limiter le nombre de personnes habilitées à modifier les données :

- niveau consultation : autorisation de consulter les données, sans les modifier ;
- niveau mise à jour : autorisation de modifier les données relatives aux personnes, fonctions et organismes, en ajouter, en supprimer ;
- niveau exploitation : autorisation de modifier, de créer et de supprimer des catégories et leurs associations aux personnes, fonctions et organismes ;
- niveau administrateur : autorisation de modifier les données relatives aux utilisateurs, d'en ajouter et d'en supprimer.

Lorsqu'un utilisateur modifie une donnée, cette information doit être enregistrée, soit sous forme du dernier utilisateur ayant modifié l'objet, soit via un historique complet.

3.7 Schéma

Suite à cette analyse, voici le modèle entité-relation qui a été réalisé.


street
number
box
zip_code
city
Address
habite
1,1
0,N
se situe
0,N
0,N
description
first_name
last_name
Person
0,N
occupe
1,1
Subfunction
1,1
est composée
0,N
Function
0,N
est composée
0,N
Organism
status
description
name
0,N
appartient à
0,N
Category
0,N
appartient à
0,N
name
type

Chapitre 4

Choix et justification des outils

4.1 Ruby on Rails

Ruby on Rails, également appelé RoR ou Rails est un framework web libre écrit en Ruby. Il suit le motif de conception Modèle-Vue-Contrôleur.

4.1.1 Philosophie

Ruby on Rails se base sur deux principes fondamentaux :

- *Ne pas se répéter* : Les éléments de l'application ne peuvent se retrouver qu'à un seul endroit. Ce principe est souvent appelé DRY (Don't repeat yourself) sur Internet,
- *Convention plutôt que Configuration* : Il est inutile de préciser des détails lorsqu'ils respectent des conventions établies par le framework. Rails exploite cela en proposant des comportements par défaut pour la plupart de ses fonctionnalités.

4.1.2 Motif de conception MVC

C'est une architecture et un modèle de conception organisant une application, ainsi que l'interface Homme-Machine, qu'il sépare en 3 parties : le modèle de données, l'interface utilisateur et la logique de contrôle.

Les relations entre les parties sont illustrées dans le graphique 4.1

Ce motif de conception appliqué à Ruby on Rails :

- Les **modèles** sont les classes assurant la gestion des données.
- Les **vues** déterminent comment sont affichées les informations à l'utilisateur.
- Les **contrôleurs** réagissent aux requêtes utilisateur et répondent généralement à l'aide des vues.

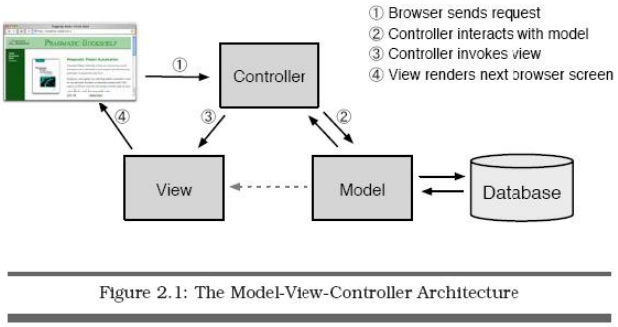


Figure 2.1: The Model-View-Controller Architecture

FIG. 4.1 – Source : Agile Web Development With Rails

4.1.3 Intérêt de Ruby on Rails pour le client

Le CSA utilise déjà Ruby on Rails pour son site Internet. Ce framework a démontré une grande flexibilité en terme de facilité de programmation et de conventions permettant de bonnes pratiques. Le code produit est de ce fait plus lisible, plus simple à reprendre par la suite et compréhensible même par un informaticien débutant dans Ruby.

Toutes ses qualités font voir en Ruby on Rails une solution permettant de réaliser des projets pérennes.

Le coté Web permet de réaliser une solution disponible via un Intranet. Cela permet une accessibilité de l'application à tous avec tout ordinateur et sous tous navigateurs. La base de données est unique et accessible via des contraintes de mots de passe par l'application. De même, toute modification est accessible directement par tous les utilisateurs.

4.2 Subversion

Subversion (en abrégé SVN) est un système de gestion de versions permettant de suivre la progression de l'application via un mécanisme de modifications et d'évolutions réversibles. Il permet de marquer facilement des versions à différentes étapes du développement, que celles-ci soient des versions de tests ou des versions majeures. Au final, Subversion permet un travail collaboratif entre personnes sur un même projet.

4.2.1 Intérêt de Subversion pour le client

Grâce à la mise en place d'un outil de gestion de version, l'intérêt est double :

- donner de bonnes pratiques à l'étudiant pour ses futurs projets ;
- avoir une source unique et centralisée du code pouvant être récupérée sur des ordinateurs différents par les personnes accréditées. Le client est donc plus actif dans l'élaboration du projet vu qu'il peut tester et voir les avancées au jour le jour.

4.3 Méthode agile

Une méthode agile est une méthode de développement informatique permettant de concevoir des logiciels en impliquant au maximum le demandeur (client), ce qui permet une grande réactivité à ses demandes. Les méthodes agiles se veulent plus pragmatiques que les méthodes traditionnelles. Elles visent la satisfaction réelle du besoin du client, et non d'un contrat établi préalablement. La notion de méthode agile est née à travers un manifeste signé par 17 personnalités (parmi lesquelles Ward Cunningham, l'inventeur du Wiki), créateurs de méthodes ou dirigeants de sociétés.

— Source : Wikipedia[1] —

4.3.1 Valeurs

La méthode prône 4 valeurs fondamentales :

- *l'équipe (« Personnes et interaction plutôt que processus et outils ») : Dans l'optique agile, l'équipe est bien plus importante que les moyens matériels ou les procédures. Il est préférable d'avoir une équipe soudée et qui communique, composée de développeurs moyens, plutôt qu'une équipe composée d'individualistes, mme brillants. La communication est une notion fondamentale ;*

- *l'application (« Logiciel fonctionnel plutôt que documentation complète ») : Il est vital que l'application fonctionne. Le reste, notamment la documentation technique, est secondaire, même si une documentation succincte et précise est utile comme moyen de communication. La documentation représente une charge de travail importante, mais peut pourtant être néfaste si elle n'est pas à jour. Il est préférable de commenter abondamment le code lui-même, et surtout de*

[1] http ://fr.wikipedia.org/wiki/Méthode_agile

transférer les compétences au sein de l'équipe (on en revient à l'importance de la communication) ;

- *la collaboration (« Collaboration avec le client plutôt que négociation de contrat ») : Le client doit être impliqué dans le développement. On ne peut se contenter de négocier un contrat au début du projet, puis de négliger les demandes du client. Le client doit collaborer avec l'équipe et fournir un feedback continu sur l'adaptation du logiciel à ses attentes ;*

- *l'acceptation du changement (« Réagir au changement plutôt que suivre un plan ») : La planification initiale et la structure du logiciel doivent être flexibles afin de permettre l'évolution de la demande du client tout au long du projet. Les premières versions du logiciel vont souvent provoquer des demandes d'évolution.*

— Source : Wikipedia[2] *—*

4.3.2 Intérêt de la méthode agile pour le client

La méthode agile permet une grande interaction entre l'étudiant et le CSA. C'est une méthode profitable à tous. En effet le client a un grand contrôle sur l'application et l'étudiant a un feedback régulier et peut poser des questions précises suite aux commentaires du CSA.

Le gain de temps qui en découle est très appréciable car nous allons ensemble vers un objectif commun.

[2] http ://fr.wikipedia.org/wiki/Méthode_agile#Valeurs

Chapitre 5

L'architecture MVC dans l'application

5.1 Modèles

5.1.1 Introduction

Le modèle est la couche qui interagit avec la base de données via :

- le traitement des données (validation avant enregistrement, création de hachage MD5[1] ou SHA-1[2] pour les mots de passe, vérification avant destruction d'un objet, ...) ;
- la description des objets (liens entre objets) ;
- la gestion de ces données et la garantie de leur intégrité ;
- des méthodes pour récupérer des données.

Les modèles ont comme nom le singulier du nom de la table et ne comportent pas d'espace. (Par exemple, le modèle « Category » aura par convention comme table « categories ».)

Si le nom de la classe a plusieurs majuscules, Rails insérera des tirets bas entre les mots pour trouver le nom de la table. Par exemple, le modèle avec la classe OrganismCategory sera liée par défaut à la table organisms_categories.

Les modèles se trouvent dans le répertoire app/models.

[1] L'algorithme MD5, pour Message Digest 5, est une fonction de hachage cryptographique très populaire, mais qui n'est plus considérée comme un algorithme sûr pour toutes les utilisations.

[2] SHA-1 (Secure Hash Algorithm) est une fonction de hachage cryptographique conçue par la National Security Agency des États-Unis et publiée par le gouvernement des États-Unis comme un standard fédéral de traitement de l'information. Elle produit un résultat (appelé « hash » ou condensat) de 160 bits.

5.1.2 Analyse de partie de modèles

5.1.2.1 Les validations

app/models/user.rb

```
1  [...]
2
3    validates_presence_of :login, :person_id
4    validates_presence_of :password, :if => :password_required?
5    validates_uniqueness_of :login
6    validates_length_of :login, :within => LOGIN_RANGE
7    validates_length_of :password, :within => PASSWORD_RANGE,
8                                   :if => :password_required?
9
10   validates_format_of :login,
11                       :with => /^[A-Z0-9_]*$/i,
12                       :message => "must contain only letters, "
13                        + "numbers, and underscores"
14
15   validates_confirmation_of :password, :if => :password_required?
16
17   [...]
```

Nous pouvons constater ici les obligations avant de créer ou d'enregistrer une nouvelle entrée à la base de données. Si l'entrée potentielle ne répond pas à ces conditions, elle est rejetée. Un message spécifique peut être renvoyé pour être affiché par la vue. Celui-ci se met alors dans le paramètre :message.

- *validates_presence_of* permet de vérifier la présence ;
- *validates_length_of* permet de vérifier la longueur (ici via une constante) ;
- *validates_format_of* permet de vérifier via le contenu.

5.1.2.2 Vérification

app/models/user.rb

```
1  [...]
2    def after_destroy
3      if User.count.zero?
4        raise "Can't delete last user"
5      end
6    end
7  [...]
```

Voici une méthode intéressante. Cette méthode est qualifiée de « méthode réservée ». Ces méthodes sont automatiquement appelées comme leur libellé l'indique avant ou après certaines commandes afin de réaliser des opérations sur les données. Par exemple, *before_validation, before_validation_on_create, after_validation, before_save, before_create, after_create, after_save*.

Pour notre cas, il s'agit d'une vérification avant suppression finale. S'il n'y a plus de profil utilisateur après suppression, un rollback est effectué signalant à l'usager que la suppression est impossible et lui envoyant un mot d'explication.

5.1.2.3 Gestion des données

app/models/cart.rb

```
1  class Cart
2    attr_reader :items
3
4    def initialize
5      @items = []
6    end
7
8    def add_item(object)
9
10     @items << object unless @items.include? object
11
12   end
13
14   def remove_item(object)
15
16     @items.delete(object)
17   end
18
19 end
```

Nous avons ici la classe concernant le panier. Le panier sert à effectuer des traitements groupés pour un ensemble d'objets similaires. Ces objets sont rassemblés dans un array. L'ajout se fait si le nouvel élément n'existe pas déjà dans l'array et le retrait se fait en appelant une méthode de la classe Array.

5.1.2.4 Description des objets

app/models/person.rb

```
1  class Person < ActiveRecord::Base
2    has_many :addresses, :as => :reference
3    has_one :address, :as => :reference, :conditions => "addresses.default = true"
4
5    has_many :emails, :as => :reference
6    has_one :email, :as => :reference, :conditions => "emails.default = true"
7
8    has_many :functions
9    has_one :function, :conditions => "functions.default_person = true"
10
11   has_and_belongs_to_many :categories, :class_name => 'PersonCategory',
12                           :join_table => 'categories_people',
13                           :association_foreign_key => 'category_id'
14 [...]
```

Cette classe fait la relation avec la table *people*. C'est le principe ORM (Object/Relational Mapping). Il existe un mapping entre la table (*people*) et la classe (*Person*).

Les déclarations *has_many*, *has_one* et *has_and_belongs_to_many* créent les méthodes pour accéder aux informations via la clef étrangère (générée automatiquement ou forcée à la main si elle ne correspond pas aux principes des conventions) de manière totalement transparente.

Nous pouvons constater qu'une personne :

- peut avoir plusieurs adresses (Le paramètre suivant fait référence à l'association polymorphique[3].) ;
- a une méthode *address* permettant d'accéder directement à l'adresse par défaut ;
- peut avoir plusieurs adresses email ;
- a une méthode *email* permettant d'accéder directement à l'adresse électronique par défaut ;
- peut appartenir à plusieurs catégories (relation N-N via une table de jointure avec « forçage » du nom de la clef étrangère).

5.2 Contrôleurs

5.2.1 Introduction

Le contrôleur orchestre l'application. C'est lui qui reçoit les évènements en provenance de l'encodage de l'utilisateur. Il interagit avec les modèles et envoie les affichages à la vue.

Toutefois, ce n'est pas tout à fait exact. En effet, l'encodage de l'utilisateur passera d'abord par le routage de l'application. Ceci permet de simplifier les URL encodées.

Prenons un exemple : http ://0.0.0.0 :3000/organisms/show/1

Celui-ci prendra d'abord en compte la première partie de l'URL. Celle-ci est celle du serveur Web. Ensuite, viendra le nom du contrôleur (ici le contrôleur *organisms*). L'action *show* sera appliquée avec l'identifiant 1.

En définitive, la fiche de l'organisme possédant l'identifiant 1 sera affichée.

Les noms de classe de contrôleurs ont également des conventions. Ceux-ci sont au pluriel (par exemple OrganismsController). Nous savons donc automatiquement que, sauf contre-indication, le contrôleur est lié à la table *organisms*.

Les contrôleurs se trouvent dans le répertoire app/controllers.

5.2.2 Analyse de parties de contrôleurs

5.2.2.1 La table de hachage params

Cette table est LE moyen de transmettre des paramètres entre les vues et le contrôleur. Une bonne partie des formulaires pour la création et l'édition regorge de paramètres transmis automatiquement afin d'enregistrer une nouvelle entrée dans la base de donnée de manière totalement transparente.

Bien entendu, certains paramètres se retrouvent dans l'URL tels le contrôleur, l'action ou l'identifiant de l'objet.

Mais rien ne nous empêche de créer nous-mêmes des paramètres (clef et valeur) à transmettre.

[3] les associations polymorphiques seront abordées plus tard

app/controllers/people_controller.rb

```
1  [...]
2    def autocomplete_add_category
3      @person = Person.find(params[:id])
4      category_names = params[:category].split(",")
5      category_names.each do |name|
6        category = PersonCategory.find_or_create_by_name(name.strip)
7        @person.categories << category unless @person.categories.include?(category)
8      end
9      if @person.save
10       redirect_to :action => 'show', :id => @person
11       flash[:notice] = 'Category was successfully added or already added.'
12     else
13       flash[:warning] = 'Category was not successfully added.'
14     end
15   end
16 [...]
```

Voici un exemple « classique » d'une action d'un contrôleur. Celle-ci ajoute une catégorie à une personne donnée via un champ d'auto-complétion[4].

La catégorie est un genre de « tagging » permettant de regrouper des personnes ayant un point commun (présent à une réunion, même occupation économique, ...).

La personne qui sera ajoutée à cette catégorie est alors recherchée. L'identifiant de cette personne, qui est le même que celui de la base de données, est alors récupéré. Nous possédons donc 2 paramètres :id et :category qui varieront selon la personne et la catégorie saisie.

Le programme vérifiera la validité de la demande. En effet, une même catégorie ne peut être ajoutée 2 fois à une même personne.

[4]Cette particularité sera abordée plus tard

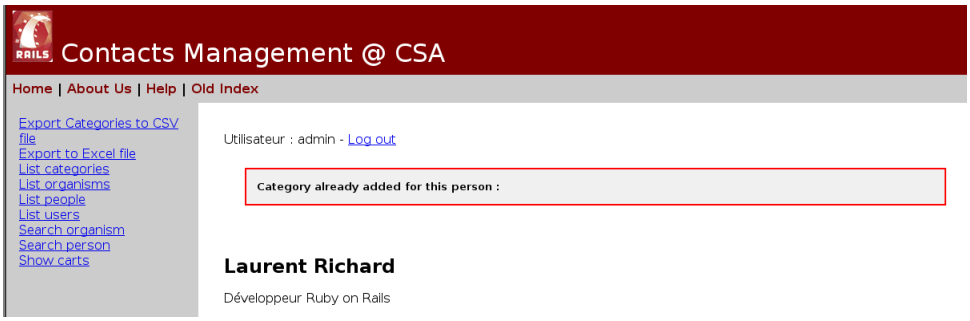


FIG. 5.1 – Résultat de l'utilisation de flash[:warning]

On aurait pu imaginer une vérification directement dans le modèle de la catégorie :

```
if params[:savecat] == '1' && params[:type] =='Person'

  Person.find(params[:id]).categories.each { |item|
    if item == Category.find(params[:cat_id])
      flash[:notice] = 'Category already existed for this person.'
      @duplicate = true

    end
    }
  if @duplicate == false then Person.find(params[:id]).categories << Category.find(params[:cat_id])
 end
end
```

Néanmoins, celle-ci s'avère moins précise et esthétique qu'une contrainte forte d'intégrité sur la table de jointure de la base de données, ainsi qu'une vérification par le contrôleur si la catégorie est déjà associée à la personne[5].

Voici donc la solution envisagée. Notons que cette ligne constitue une inclusion SQL dans un fichier de migration Rails[6].

db/migrate/006_create_categories.rb

```
1    execute "ALTER TABLE categories_people ADD CONSTRAINT
2                        person_category_id UNIQUE (person_id, category_id);"
```

Si aucune erreur n'apparaît, c'est que l'ajout s'est bien déroulé ou que la catégorie existait déjà. Si, par contre,la catégorie n'a pas pu être ajoutée, il sera nécessaire de renvoyer cette information à l'utilisateur via le flash[:warning] qui fera apparaître un cadre avec un message (ici spécifié mais pouvant être standardisé selon l'erreur).

[5] Vérifiée par le test *@person.categories.include ?(category)*

[6] Les fichiers de migrations Rails permettent de s'affranchir des scripts SQL de création et de mise à jour et, mieux encore, de versionner les schéma DB en parallèle avec le reste du code source

5.2.2.2 Les sessions

Par défaut, Ruby on Rails manipule les identifiants de session[7] de manière très correcte. En cas d'emploi avec la configuration par défaut, Ruby on Rails[8] :

- crée un cookie utilisant un hachage MD5 des données avec des quantités acceptables d'entropie[9] ;
- semble éviter des attaques de fixation de session en n'utilisant les identifiants de session qu'au niveau des cookies et non dans les liens ;
- stocke l'état de la session (par exemple, @session [:user]) disponible sur le serveur où il est trouvé par identification de session sans être sujet à la manipulation par l'utilisateur.

En plus de l'usage pour des fins d'authentification, j'ai fait usage de ces possibilités pour créer les paniers de personnes et d'organismes.

Ces paniers sont, comme pour les sites de commerce en ligne, utilisés pour rassembler des éléments identiques et leur affecter un traitement commun. Ici, on ne vend rien, mais quand il est question d'affecter une catégorie à 20-30 personnes, il est plus simple d'appliquer cela à un groupe préétabli que d'affecter la catégorie manuellement à chacune des personnes ou organismes.
Dans l'avenir, il sera également question d'utiliser ces paniers pour générer des envois de mails groupés ou des mailings papier.

app/controllers/people_controller.rb

```
1  [...]
2     def add_to_cart
3       begin
4         @person = Person.find(params[:id])
5       rescue ActiveRecord::RecordNotFound
6         logger.error("Attempt to access invalid person #{params[:id]}")
7         flash[:notice] = "Invalid person"
8         redirect_to :action => :list
9       else
10        @cart= find_cart_person
11        @cart.add_item(@person)
12      end
13      redirect_to :controller => 'carts', :action => 'show'
14    end
15
16    private
17    def find_cart_person
18      session[:cart_person] ||= Cart.new
19    end
20 [...]
```

Nous pouvons constater qu'afin de pouvoir faire des paniers propres à l'utilisateur, la création et le transfert des informations contenues dans le panier se réalise via la session. Nous créons donc un emplacement destiné à :cart_person dans la session s'il n'existe pas (ligne 18). Une variable interne est utilisée pour gérer le panier plus facilement.

Pour vider le panier, on annule le contenu de la session (session[:cart_person] = nil). Simple et efficace.

Le panier n'ayant pas d'existence dans la base de données, une classe Cart non liée à la base de données et ayant ses méthodes définies dans un modèle a été créé.

[7] L'identification de session est un hachage MD5 basé sur le temps, un nombre aléatoire et une chaîne constante. Cette routine est employée en interne de manière automatique afin de générer les identifiants session.

[8] http ://wiki.rubyonrails.org

[9] On nomme entropie la quantité physique qui mesure le degré de désordre d'un système. On peut parler de l'entropie d'une suite de bits et mesurer son degré de désordre : plus cette suite de bits se rapproche d'une suite aléatoire, plus son entropie sera grande.

app/models/cart.rb

```
1  class Cart
2    attr_reader :items
3
4    def initialize
5      @items = []
6    end
7
8    def add_item(object)
9
10     @items << object unless @items.include? object
11
12   end
13
14   def remove_item(object)
15
16     @items.delete(object)
17   end
18
19 end
```

5.2.2.3 Les redirections

Sans le savoir, nous avons introduit la partie suivante de l'analyse du programme. En effet, une fois les traitements de l'action du contrôleur terminés, il est temps de procéder à un affichage ou d'effectuer une redirection.

Tout en essayant de rester assez bref, nous dirons que nous avons deux types d'affichages.

- :redirect_to
- :render

Redirect_to sert pour une nouvelle requête. Il est donc utilisé principalement pour passer d'un contrôleur à un autre ou lors d'une nouvelle recherche ou affichage. Cela génère un code HTTP 302[10].

Render sera dès lors utilisé pour les changements ou affichages ne réclamant pas de nouvelle requête. Les cas les plus fréquents sont ceux où les informations encodées dans les formulaires de création ou d'édition ne sont pas validées par le modèle. On obtient donc la même page avec, éventuellement, un message d'erreur. Les informations encodées sont maintenues dans le formulaire pour modification immédiate sans devoir ressaisir les données. Cela générera un code HTTP 200[11].

5.3 Vues

5.3.1 Introduction

Nous avons considéré les modèles et les contrôleurs, les actions du contrôleur faisant aussi partie de l'URL à utiliser, et le choix de l'action par celui-ci. Nous avons également vu comment rediriger l'information entre les contrôleurs ou à l'intérieur d'un même contrôleur.

Venons en à présent au résultat via les « templates ». En effet, le contenu n'est pas affiché directement mais bien généré.

Les vues se trouvent par défaut sous le répertoire app/views.

Prenons en compte le fait que les pages sont générées pour afficher l'information. Nous n'avons pas d'intérêt à avoir des pages statiques alors qu'il est possible de transmettre des variables entre les contrôleurs et les vues, mais aussi de les utiliser directement pour afficher ce que l'on souhaite (le nom

[10] 302 - Moved Temporarily - Document déplacé de façon temporaire
[11] 200 - OK - Requête traitée avec succès

de notre utilisateur par exemple). C'est la raison pour laquelle les vues ne sont pas des fichiers avec une extension .html mais des fichiers .rhtml.

Bien entendu, les variables sont définies dans les actions portant le même nom que la vue par défaut.

En dehors du code HTML habituel permettant de définir le style de l'affichage, il existe deux manières d'inclure du code Ruby.

La première est d'appeler le résultat d'une commande. <%= @category.name %>, par exemple, va afficher le nom de la catégorie. Il ne s'agit pas vraiment d'une commande à proprement parler. En fait, il s'agit plutôt de la valeur renvoyée par la méthode appelée ou la variable spécifiée à laquelle on tente « d'appliquer » la méthode *.to_s* pour en faire une simple chaîne de caractères.

<
La deuxième est d'exécuter du code.

```
<% 3.times do %>
Hello World!<br/>
<% end %>
```

Ce code affichera trois fois le message « Hello World ! ».

Dès lors, rien ne nous empêche de récupérer des informations données dans l'URL par l'utilisateur tels le contrôleur, l'action, l'identifiant ou même un autre paramètre.

Dans un souci de sécurité et afin d'éviter des attaques de type XSS (Cross Site Scripting), Ruby on Rails propose une méthode utilisable dans les fichiers de vues. La méthode *h()* permet tout simplement de transformer en entités HTML tous les caractères préjudiciables au contenu Web traditionnel, évitant ainsi des problèmes de modifications du rendu de la page ou, pire, l'utilisation de Javascript malveillant.

Nous aurons l'occasion d'en reparler par la suite.

5.3.2 Analyse de partie de vues

Prenons en exemple la création d'une personne.

Tout part de l'action *new* du contrôleur PeopleController.

app/controllers/people_controller.rb

```
1  [...]
2    def new
3      unless params[:organism].blank?
4        @organism = Organism.find(params[:organism])
5      end
6      @person = Person.new
7      create_myfunctions
8      @myfunctions.first.organism = @organism
9    end
10
11     private
12
13     def create_myfunctions
14        @myfunctions = Array.new
15        3.times do
16          @myfunctions << Function.new
17        end
18     end
19 [...]
```

Dans cette action du contrôleur, rien de très particulier. On y définit que @person désigne une nouvelle personne et il est fait appel à une procédure privée où il est défini que @myfunction est un Array de 3 nouvelles fonctions.

Vient ensuite le passage implicite à la vue (new.rhtml).

app/views/people/new.rhtml

```
1  <h1>New person</h1>
2
3  <% form_tag :action => 'create' do %>
4    <%= render :partial => 'form' %>
5
6  <br />
7
8    <%= submit_tag "Create" %>
9  <% end %>
10
11 <%= link_to 'Back', :action => 'list' %>
```

Étrangement, le code nécessaire pour l'affichage de la vue semble restreint[12]. En effet, nous remarquons une balise de code un peu spéciale <%= render :partial => 'form' %>. La notion de partial sera abordée plus en détails dans le chapitre suivant consacré au principe DRY. (En fait, on appelle une « sous-vue » nommée *_form* (le nom entre guillemets précédé d'un tiret bas).) Notons l'usage du render (cfr la section 5.2.2.3). Que nous apprend cette nouvelle vue ?

app/views/people/_form.rhtml

```
1  <%= error_messages_for 'person' %>
2
3  <p><label for="person_last_name">Last name</label><br/>
4  <%= text_field 'person', 'last_name' %></p>
5
6  <p><label for="person_first_name">First name</label><br/>
7  <%= text_field 'person', 'first_name' %></p>
8
9  <h3>Sex</h3>
10 <%= radio_button 'person', 'sex', "M" %> Male <br />
11 <%= radio_button 'person', 'sex', "F" %> Female <br />
12
13 <p><label for="person_description">Notes</label><br/>
14 <%= text_area 'person', 'description', 'rows' => 5 %></p>
15
16 <p><label for="person_phone_number">Phone number</label><br/>
17 <%= text_field 'person', 'phone_number' %></p>
18
19 <p><label for="person_mobile_number">Mobile number</label><br/>
20 <%= text_field 'person', 'mobile_number' %></p>
21
22
23 <table border=1>
24   <tr>
25    <td><label for="function_name">Function Name</label></td>
26    <td><label for="function_organism_id">Organism</label></td>
27
28   </tr>
29
30   <% @myfunctions.each do |@item| %>
31   <tr>
```

[12] Pour information, la balise <% form_tag :action => 'create' do %> nous crée un formulaire qui sera envoyé à l'action *create* lorsqu'on appuiera sur le bouton <%= submit_tag "Create" %>.

```
32      <td><%= text_field_tag("item[title][]", @item.title, "size" => "50") %></td>
33      <td><select id="item_organism_id" name="item[organism_id][]">
34      <option value=""> </option>
35      <%= options_from_collection_for_select(Organism.find(:all), "id", "name",
36                  selected_value = @item.organism_id) %>
37          </select></td>
38    </tr>
39    <% end %>
40  </table>
```

Globalement, ce formulaire nous montre le lien étroit entre le label et le champ du formulaire. Les champs seront transmis par paramètres vers la base de données via le contrôleur et le modèle. Il est donc assez simple d'intercepter des informations en cas de besoin dans le contrôleur ou le modèle. Cela pourrait être, par exemple, le cas pour un formulaire d'utilisateur avec mot de passe. Vu que le mot de passe n'est jamais écrit en clair dans la base de données, le hachage MD5 ou SHA-1 sera généré.

La dernière partie est tout à fait digne d'intérêt. Un tableau permettant d'ajouter les fonctions de la personne est créé. La deuxième colonne permet de sélectionner dans un menu déroulant un organisme pour lequel la personne exercerait cette fonction.

Occupons-nous à présent de l'action *create* du contrôleur vu que c'est cette action qui permettra réellement de transférer ou non les informations au modèle.

app/controllers/people_controller.rb

```
1  [...]
2    def create
3      @person = Person.new(params[:person])
4      @person.updated_by = @current_user
5      if @person.save
6        filling_myfunctions
7      end
8      if @person.save
9        flash[:notice] = 'Person was successfully created.'
10       redirect_to :action => 'list'
11     else
12       flash[:notice] = 'Problem occured during the creation process'
13       create_myfunctions
14       render :action => 'new'
15     end
16   end
17
18    private
19    def filling_myfunctions
20         @new_items = Array.new
21
22         for i in 0...params[:item]['title'].length
23           unless params[:item]['title'][i].blank?
24             organism = params[:item]['organism_id'][i].blank? ? nil :
25         organism = Organism.find(params[:item]['organism_id'][i])
26             @new_items << Function.new(:organism => organism,
27         :person => @person, :title => params[:item]['title'][i])
28           end
29         end
30         @person.functions(true)
31         for item in @new_items
32           item.save
33         end
34    end
35 [...]
```

La création se fait ici en 2 étapes. Les informations de la personnes sont injectées dans une nouvelle instance de la classe *Person*. Si la sauvegarde des attributs de la personne se réalise avec succès, le système passera à la sauvegarde des fonctions.

Chapitre 6

Mise en application du principe DRY

6.1 Introduction au DRY

DRY ou *Don't Repeat Yourself* est une philosophie de processus visant à réduire la duplication, en particulier dans le calcul. La philosophie souligne que l'information ne devrait pas être reproduite, parce que la duplication augmente la difficulté du changement, peut diminuer la clarté, et peut générer des contradiction.

Néanmoins, il ne faut pas tomber dans l'autre extrême. L'information peut être dupliquée si nécessaire mais il faut au maximum minimiser les doublons en visant toujours l'idée de maintenabilité, lisibilité et réutilisabilité du code. Un code réutilisable est du travail en moins plus tard.

Un code concis est un code où les modifications se feront plus rapidement et seront donc mieux acceptées par le développeur.

Rails implémente plusieurs fonctionnalités permettant au développeur de ne pas se répéter. Ainsi les éventuelles modifications ne se feront qu'à un endroit. L'appréhension de la « confrontation » avec les utilisateurs disparaît et les rencontres avec eux deviennent des séances d'exploration des possibilités.

Nous ne pensons pas que le principe DRY augmente les performances en diminuant le calcul. L'inlining[1] est en général plus performant que l'appel de fonctions (génériques ou non). Par contre, cette méthode aide grandement à la maintenance, à la lecture et à la compréhension du code.

6.1.1 Les Helpers

Les helpers sont des assistants dont la principale tâche est de limiter un maximum le code Ruby à écrire dans les formats des vues (conformément au principe DRY). Ainsi le format sera essentiellement composé de HTML (ou de XML). Rails est livré avec un grand nombres d'assistants facilitant :

- le formatage des données ;
- les liens entres les pages ;
- la pagination ;
- la gestion des erreurs ;
- la création de formulaires ;
- et encore d'autres . . .

Rien empêche d'écrire ses propres helpers afin faciliter la concision de son code.

6.1.2 Mise en page

La règle « DRY » est également appliquée à la composition globale des pages affichées dans une application Ruby on Rails. C'est la suite logique et le complément de la programmation avec des CSS.

[1] http ://en.wikipedia.org/wiki/Inlining

Il est courant dans la création de sites Internet d'utiliser, par exemple, un en-tête, un pied de page et un menu identique sur l'ensemble des pages. Ce sont ces éléments qui seront les premiers à bénéficier des layouts. Encore une fois, c'est un gain de temps et un investissement afin d'éviter la duplication du code qui peut devenir préjudiciable à terme.

Rails propose de centraliser ces différentes mises en page dans le répertoire *layouts* (situé à l'intérieur du répertoire *app/views*) et de les invoquer à la demande plutôt que de répéter du code sur toutes les vues.

6.1.3 Pages partielles ou *partials*

Conformément à la règle « DRY » (Don't Repeat Yourself), le développeur peut créer des portions de vue qui seront intégrées à d'autres lors de l'affichage.

Dans l'exemple vu précédemment, le fichier *_form.rhtml* contient un formulaire de saisie. Ce formulaire étant commun aux deux actions de création et d'édition, il a été extrait des vues et seule une référence à ce fichier est présente dans *new.rhtml* et *edit.rhtml*. Ainsi, si le développeur souhaite modifier le contenu du formulaire, il ne le fera qu'une fois au lieu de deux.

6.2 Principes DRY appliqués au projet

6.2.1 Les Helpers

Cette partie n'a pas été très approfondie en pratique. Il est vrai que des processus pour lesquels l'utilisation des helpers aurait été utile ne sont pas apparus au premier abord.

6.2.2 Réécriture du code afin de minimiser les tests ou la duplication du code

Certains vous parlerons d'esthétisme, d'autres d'art de coder, je préfère parler de minimiser les tests et la duplication du code.

J'ai tendance à ne pas aimer les tests conditionnels mais à créer deux fonctions et à les nommer par rapport à ce qu'elles font.

Ce fut le cas par exemple ici.

app/views/carts/_cart_item.rhtml

```
1 <tr>
2     <td> <%= h(cart_item.full_name) %>
3     <td><%= link_to image_tag("/icons/delete.png", :border=>0, :alt => "Delete"),
4           { :action => "delete_item_#{h(cart_item.class.to_s.downcase)}",
5           :id => cart_item } %></td>
6  </tr>
```

Ceci est le code d'un partial appelé par la vue *show* du panier. Cette vue affiche donc l'ensemble du contenu de chaque panier avec un listing. Le partial est donc appelé avec une collection qui est celle des éléments du panier.

Chaque élément est donc, conformément au partiel, affiché sur base de son nom complet (prénom et nom pour une personne ou nom et statut pour un organisme). Il est suivi par un lien en forme de bouton dans le cas où l'utilisateur souhaiterait supprimer cette personne ou cet organisme du panier.

Le contrôleur appelé est le même que le répertoire de la vue à savoir *carts*. L'action, par contre, va varier selon la classe de l'objet listé (*delete_item_person* pour les personnes et *delete_item_organism* pour les organismes)

app/controllers/carts_controller.rb

```
1  [...]
2    def delete_item_person
3      @cart = find_cart_person
4      item_to_remove = Person.find(params[:id])
5      @cart.remove_item(item_to_remove)
6      flash[:notice] = "Item removed"
7      redirect_to :action => 'show'
8    end
9
10   def delete_item_organism
11     @cart = find_cart_organism
12     item_to_remove = Organism.find(params[:id])
13     @cart.remove_item(item_to_remove)
14     flash[:notice] = "Item removed"
15     redirect_to :action => 'show'
16   end
17 [...]
```

Les deux fonctions affectent donc leurs variables différemment. On pourra, pour aller encore plus loin dans la factorisation du code, regrouper le code commun dans une troisième fonction. Factoriser du code en fonctions n'apporte jamais de gain de performance, mais bien un gain de lisiblité et de compréhension du code.

6.2.3 Les partials

Sur le conseil du client, un effort particulier a été fait pour appliquer au maximum les principes DRY.

Cela s'est surtout marqué sur les pages partielles. Même si une partie d'entre elles a été générée automatiquement sur base du schéma de la base de données, de nombreux aspects ont dû être peaufinés à la main.

Il n'était pas envisageable de laisser les utilisateurs modifier sans limites les données indispensables aux associations polymorphiques. Cela aurait été, il est vrai, suicidaire. Le problème est venu du fait que ces données faisaient partie intégrante de celles dont le modèle avait besoin pour la validation. Dès lors, on ne pouvait se passer de leur présence. Une particularité de Rails est la possibilité de masquer certains champs. Ceux-ci sont alors non visibles et non éditables : nous avions notre solution. Les champs de la base de données se terminant par *_at* comme *created_at* ou *update_at* sont mis à jour automatiquement. Ceux-là sont donc supprimables de la vue car gérés automatiquement, ce qui est encore un gain de temps. Il ne nous restait quasiment plus qu'à ne pas oublier de modifier *updated_by* directement en codant la ligne dans le contrôleur.

Cependant, les partials prennent tout leur sens via les listes et les sous-listes. Il suffit de regarder le nombre de fichiers dans *app/views/shared* pour s'en rendre compte. Mais prenons plutôt l'exemple de la vue *show* du panier afin de commencer.

app/views/carts/show.rhtml

```
1
2  <div class="cart-title"> Your CSA cart for people</div>
3
4
5  <br />
6  <table>
7  <%= render(:partial =>"cart_item", :collection => @cart_person.items) %>
8  </table>
9  <br />
10 <% @categories = Category.find_all_by_type('PersonCategory')%>
11
12 <%= render(:partial =>"add_category") %>
13 <br />
14 <%= button_to "Empty cart", :action => 'empty_cart_person' %>
15 <%= button_to "Create Excel file", :controller => 'site',
16           :action => 'export_people_to_excel' %>
17 <br />
18  [...]
```

Voici le code qui affiche le contenu du panier de personne. Outre le titre, nous sommes directement confrontés à un partial auquel on envoie un paramètre :collection. En fait, on envoie bien l'ensemble du contenu du panier au partial. Automatiquement, sans aucune autre indication, Rails va parcourir cet array et appliquer le partial à chacun des éléments de l'array.

Notons aussi que l'appel du partial se réalise sans le tiret bas avant le nom.

app/views/carts/_cart_item.rhtml

```
1  <tr>
2     <td> <%= h(cart_item.full_name) %>
3     <td><%= link_to image_tag("/icons/delete.png", :border=>0, :alt => "Delete"),
4                   { :action => "delete_item_#{h(cart_item.class.to_s.downcase)}",
5                   :id => cart_item } %></td>
6   </tr>
```

Encore une fois, le code se réduit à quelques lignes. Comme vu dans le point précédent, une méthode de même nom a été créée dans chaque classe (*Person* et *Organism*) ce qui fait que nous avons une même méthode d'apparence qui donnera des résultats différents selon la classe. Cela s'appelle également du *Duck Typing*[2].

Chaque élément de la collection est représenté par le nom du partial sans le tiret bas de début de fichier. (donc ici, cart_item).

Nous avons abordé les partials utilisés par un seul contrôleur. Néanmoins, si nous prenons le cas d'une personne et d'un organisme, ils ont pas mal de points communs :

1. ils possèdent une ou plusieurs adresses ;
2. ils possèdent des adresses mails ;
3. ils peuvent être inclus dans des catégories ;
4. ...

Il devient donc utile de centraliser les partials vu que les informations sont les mêmes (avec toutefois différents types de références).

Voyons donc comment ce point a été géré. Nous prendrons l'exemple d'une personne.

[2] http ://en.wikipedia.org/wiki/Duck_typing

app/controllers/people_controller.rb

```
1   def show
2     begin
3       @person = Person.find(params[:id])
4     rescue ActiveRecord::RecordNotFound
5       logger.error("Attempt to access invalid person #{params[:id]}")
6       flash[:notice] = "Invalid person"
7       redirect_to :action => :list
8     else
9       @categories_class = PersonCategory
10      @emails = @person.emails
11      @addresses = @person.addresses
12      @entity = @person
13      @entity_controller = 'people'
14    end
15  end
```

Nous remarquons qu'il y a plus de variables définies. Cela nous servira plus tard. L'important ici est la variable *@entity* qui pointe sur la variable *@person* qui n'est jamais que la personne sélectionnée pour être affichée.

Partons à présent vers la vue associée, soit *show.rtml*.

app/views/people/show.rhtml

```
1  <h1><%= @person.full_name %> <%= "(#{@person.gender})" %></h1>
2  <%= link_to 'Edit', :action => 'edit', :id => @person %>
3  <p><%= @person.description unless @person.description.blank? %></p>
4  <%= render(:partial => "show_function" , :collection => @person.functions) %>
5
6  <%= render(:partial => "shared/address") unless @person.addresses.empty? %><p />
7  <%= link_to 'New address', {:controller => 'addresses', :action => 'new',
8          :id => @person.id, :reference => @person.class} %>
9
10 <%= render(:partial => "shared/email") unless @person.emails.empty? %><p />
11 <%= link_to 'New email', {:controller => 'emails', :action => 'new',
12         :id => @person.id, :reference => @person.class} %>
13
14
15 <%= render(:partial => "shared/category") %><p />
16 <%= render(:partial => "shared/add_category") %>
17 <%= link_to 'Back', :action => 'list' %>
```

L'important ici est l'utilisation de partials qui ne sont plus sous le répertoire réservé au contrôleur *people*. Outre les vérifications d'affichage en cas de non-présence de données comme l'email ou une adresse, nous allons nous concentrer sur l'affichage d'une seule donnée, par exemple les adresses emails.

Contrairement à notre introduction sur les partials, aucune collection ni aucun objet ne sont transmis au partial via le paramètre *:collection ou :object*. Les autres variables sont toujours disponibles pour utilisation ultérieure.

app/views/shared/_email.rhtml

```
1  <h3>Email</h3>
2  <table>
3      <% if !@entity.email.nil? %>
4            <tr>
5              <td><%=h @entity.email.address %></td>
6              <td><%= link_to image_tag("/icons/application_edit.png", :border=>0, :alt => "Edit"),
7          {:controller => 'emails', :action => 'edit', :id => @entity.email.id} %> </td>
8              </tr>
9       <% end %>
10 </table>
11 <% unless @entity.emails.size < 2 %>
12 <h4>Others Email Addresses :</h4>
13 <table>
14 <%= render(:partial => "shared/line_email" , :collection => @entity.emails) %>
15 </table>
16 <% end %>
```

Voici donc un partial « généraliste ». Il a été entièrement codé sans génération automatique, ce qui serait impossible à un tel niveau d'abstraction. Ici, nous travaillons avec les variables *@entity*.

Rappelez-vous que dans l'introduction, nous avions remarqué que, pour désigner chaque élément de la collection, nous utilisions le nom du partial sans le tiret bas. Techniquement, nous aurions pu le faire si nous avions transmis un objet au partial. Pourquoi ne pas l'avoir fait ? Il me paraît moins compréhensible et lisible d'écrire *email.email* que d'écrire @entity.email. C'est une alternative par rapport à la convention générale afin d'obtenir un code plus lisible et donc plus facilement gérable.

Nous remarquons que rien ne nous empêche d'appeler un partial dans un partial. Celui-ci nous affichera les autres adresses email disponibles.

Chapitre 7

Particularités intéressantes du framework

7.1 Associations polymorphiques

En Rails, une association polymorphique est une association qui lie des objets de différents types. Le postulat est que tous ces objets partagent des caractéristiques communes mais qu'ils auront des représentations différentes.

Soyons plus pratiques et penchons-nous sur un exemple du projet : l'adresse.

Une personne et un organisme ont chacun une ou plusieurs adresses. De manière simpliste, les tables auraient pu être créées de la manière suivante :

```
create_table :people, :force => true do |t|
  t.column :last_name, :text
  t.column :first_name, :text
end
create_table :organisms, :force => true do |t|
  t.column :name, :text
end
```

Ensuite, les modèles se présenteraient de cette façon :

```
# CECI NE MARCHE PAS
class Person < ActiveRecord::Base
  has_many :addresses
end
class Organism < ActiveRecord::Base
  has_many :addresses
end
```

Malheureusement, un tel schéma ne peut fonctionner. Quand nous codons *has_many :address* dans un modèle, cela signifie que la table *addresses* possède une clef étrangère qui renvoie à notre table. Mais, dans le cas présent, nous avons deux tables nécessitant chacune de posséder une clef dans la table *addresses*. Nous ne pouvons donc pas faire en sorte qu'une même clef étrangère renvoie à ces deux tables ... à moins d'utiliser les associations polymorphiques.

La solution est d'utiliser deux colonnes supplémentaires dans la table *addresses* pour la clef étrangère. La première contiendra l'identifiant de l'enregistrement cible et la seconde indiquera à Active Record dans quel modèle de ressource se trouve cette clef.

Nommons *reference* la clef étrangère des adresses. Nous devrons créer deux colonnes : *reference_id* et *reference_type*.

Voici la migration réelle d'une adresse :

db/migrate/005_create_addresses.rb

```
1  class CreateAddresses < ActiveRecord::Migration
2    def self.up
3      create_table :addresses, :force => true do |t|
4        t.column :street,             :string,  :null => false
5        t.column :number,           :integer, :null => false
6        t.column :box,            :integer
7        t.column :zip_code,     :integer, :null => false
8        t.column :city,         :string,  :null => false
9        t.column :reference_id,    :integer, :null => false
10       t.column :reference_type, :string,  :null => false
11       t.column :default,      :boolean, {:default => false}
12       t.column :start_on,       :date
13       t.column :end_on,         :date
14       t.column :created_on,       :timestamp
15       t.column :updated_on,       :timestamp
16       t.column :updated_by,       :integer
17
18       # Avoiding Race Conditions with Optimistic Locking
19       t.column :lock_version, :integer, {:default => 0}
20
21     end
22   end
23
24   def self.down
25     drop_table :addresses
26   end
27 end
```

Nous pouvons maintenant créer les modèles.

app/models/address.rb

```
1 class Address < ActiveRecord::Base
2   belongs_to :reference, :polymorphic => true
3   [...]
```

app/models/person.rb

```
1 class Person < ActiveRecord::Base
2   has_many :addresses, :as => :reference
3   [...]
```

app/models/organism.rb

```
1  class Organism < ActiveRecord::Base
2   has_many :addresses, :as => :reference
3   [...]
```

L'association polymorphique utilise l'option *:as* de *has_many* indiquant que l'association entre une adresse et une personne, ainsi que l'association entre une adresse et un organisme sont polymorphiques. Elles utiliseront l'attribut *resource* dans la table des adresses.

Avec quelques éléments ajoutés dans notre base de données, cela donne (affichage réduit pour l'occasion).

mysql> select * from addresses;

```
+----+------------------------+--------+------+----------+-----------+--------------+----------------+
| id | street                 | number | box  | zip_code | city      | reference_id | reference_type |
+----+------------------------+--------+------+----------+-----------+--------------+----------------+
|  1 | rue Jean Chapelié      |     35 | NULL |     1050 | Bruxelles |            1 | Organism       |
|  2 | Avenue Georgin         |      1 | NULL |     1030 | Bruxelles |            2 | Organism       |
|  3 | Rue Milcamps           |     59 | NULL |     1030 | Bruxelles |            3 | Person         |
|  4 | Rue du Moulin          |      1 | NULL |     4020 | Liège     |            5 | Organism       |
|  5 | Rue Puits-en-Sock      |     74 | NULL |     4020 | Liège     |            5 | Organism       |
|  6 | Rue Grétry             |     30 | NULL |     4020 | Liège     |            5 | Organism       |
|  7 | Rue Saint Léonard      |    236 | NULL |     4000 | Liège     |            5 | Organism       |
|  8 | Rue des Prémontrés     |      4 | NULL |     4000 | Liège     |            5 | Organism       |
|  9 | Rue Sainte Walburge    |      7 | NULL |     4000 | Liège     |            5 | Organism       |
| 10 | Place Xavier Neujean   |      9 | NULL |     4000 | Liège     |            5 | Organism       |
| 11 | Boulevard Frère-Orban  |      7 |   81 |     4000 | Liège     |            4 | Person         |
+----+------------------------+--------+------+----------+-----------+--------------+----------------+
11 rows in set (0.00 sec)
```

```
mysql> select * from people;
+----+-----------+------------+-------------+------+--------------+---------------+
| id | last_name | first_name | description | sex  | phone_number | mobile_number |
+----+-----------+------------+-------------+------+--------------+---------------+
|  1 | Janssen   | Marc       | NULL        | NULL | NULL         | NULL          |
|  2 | Delusinne | Philippe   | NULL        | NULL | NULL         | NULL          |
|  3 | Houtmans  | Pierre     | NULL        | NULL | NULL         | NULL          |
|  4 | Richard   | Laurent    | NULL        | NULL | NULL         | NULL          |
|  5 | Dubuisson | Bernard    | NULL        | NULL | NULL         | NULL          |
+----+-----------+------------+-------------+------+--------------+---------------+
5 rows in set (0.01 sec)
```

```
mysql> select * from organisms;
+----+-------------------------------------+--------+----------------------------+
| id | name                                | status | description                |
+----+-------------------------------------+--------+----------------------------+
|  1 | Conseil supérieur de l'audiovisuel  | NULL   | NULL                       |
|  2 | TVi                                 | S.A.   | Editeur du service RTL-TVi |
|  3 | RTBF                                | NULL   | NULL                       |
|  4 | Belgacom                            | S.A.   | NULL                       |
|  5 | Fortis                              | S.A.   | NULL                       |
+----+-------------------------------------+--------+----------------------------+
6 rows in set (0.01 sec)
```

Le *reference_id* est bien l'identifiant de la table *organisms* ou *people* selon le cas.

7.2 AJAX et l'auto-complétion

Avec l'avènement des techniques Web 2.0, AJAX devient une incontournable méthode de programmation Web.

AJAX est une technologie extrêmement intéressante à mettre en place puisqu'elle permet une interaction forte avec le client. L'idée est de pouvoir mettre à jour des éléments de la page sans avoir à recharger la page complète.

Qui dit AJAX implique Javascript. Cela, Rails va nous permettre de le faire sans trop de difficultés.

```
<%= javascript_include_tag :defaults %>
```

Il ne nous reste plus qu'à coder.

La première partie d'un développement orienté AJAX a été de créer et rechercher des catégories automatiquement, tant pour les personnes que pour les organismes. Toutefois, pour des raisons d'organisation du code et de temps, les paniers ne peuvent pas encore utiliser AJAX pour les procédures.

Nous devions donc réaliser de l'auto-complétion. Nous avons déjà abordé les bribes de ce qui a été fait via les contrôleurs.

app/views/shared_add_category.rhtml

```
1 <%= form_tag :controller => @entity_controller, :action => 'autocomplete_add_category',
2         :id => @entity %>
3 <h4>Add to a category</h4>
4 <%= text_field_tag 'category', {}, :size => 50 %><p>
5 <div class="auto_complete" id="category_auto_complete"></div>
6 <%= auto_complete_field :category, :url=>{:controller => 'categories',
7         :action=>'autocomplete_category',
8  :category_class => @entity_controller}, :tokens => ',' %>
```

app/controllers/people_controller.rb

```
1  [...]
2    def autocomplete_add_category
3      @organism = Organism.find(params[:id])
4      category_names = params[:category].split(",")
5      category_names.each do |name|
6        category = OrganismCategory.find_or_create_by_name(name.strip)
7        @organism.categories << category unless @organism.categories.include?(category)
8      end
9      if @organism.save
10       redirect_to :action => 'show', :id => @organism
11       flash[:notice] = 'Category was successfully added.'
12     else
13       flash[:warning] = 'Category was not successfully added.'
14     end
15   end
16 [...]
```

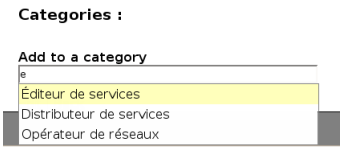


FIG. 7.1 – Résultat de l'auto-complétion

Assez simplement, nous constatons, grâce à l'action du contrôleur et surtout grâce à la méthode *OrganismCategory.find_or_create_by_name*, qu'une recherche dynamique sera effectuée et que la réponse sera ajoutée en tant que catégorie de l'organisme. Au cas où la catégorie n'existerait pas, elle sera créée automatiquement.

7.3 Gestion des tests

7.3.1 Pourquoi tester ?

Tester l'application, quoi de plus naturel en soi ? Mais est-ce toujours vrai ? De plus en plus, tout doit aller très vite. Les projets doivent être prêt pour la veille. Des applications toujours plus performantes ne cessent d'apparaître sur le marché compétitif des biens et services informatiques.

Professionnellement, réaliser des tests se révèle souvent être l'exception ou, du moins, seuls des tests basiques sont envisagés. Quels sont les risques de ne pas tester son programme ?

- Comportement aléatoire suite à une action non prévue de l'utilisateur ;
- Faille de sécurité ;
- Traitement erroné non détecté ;
- ...

Cependant, les tests sont le premier rempart pour assurer la fiabilité du programme ainsi qu'une aide non négligeable pour le développeur. Ils permettent de :

- valider les méthodes des classes et les résultats de celles-ci ;
- vérifier qu'une factorisation d'une partie de code donne toujours bien les bons résultats ;
- circonscrire le problème en cas de bug ;
- aider le développeur durant le développement en lui donnant de bonnes pratiques ;
- ...

Les méthodes actuelles fonctionnent en général sur ces principes. « Agile » en premier va plus loin en reniant le principe de développement via le modèle en cascade pour insérer les tests directement dans les itérations. Parmi ces méthodes agiles, on retrouve même des développements orientés tests.

Ces principes favorisent une approche inversée. Les tests sont préalables à la rédaction du code. On écrit donc le code qui satisfera aux tests et non plus les tests qui satisferont au code.

Itérations à suivre :

- Écrire un test ;
- Lancer les tests et voir le nouveau échouer ;
- Écrire le code ;
- Relancer les tests et voir le résultat ;
- Modifier le code si l'échec du test perdure ;
- Relancer les tests et voir le résultat ;
- Nettoyer le code en cas de réussites aux tests ;
- Relancer les tests pour valider le tout.

Cette approche a plusieurs conséquences sur les développeurs. Cela améliore les principes KISS [1] ou YAGNI[2] en mettant l'objectif à atteindre sur le succès des tests plutôt que de laisser le développeur créer du code brouillon comme c'est le cas via d'autres méthodes.

7.3.2 Les tests ... sous Ruby on Rails ?

La partie suivante se devait d'être consacrée aux tests sous Ruby on Rails. S'il est bien un framework qui met en avant les tests, c'est bien celui-là. De base, toute la panoplie des tests est disponible. Il n'est pas nécessaire de faire des ajouts. Lors de la création automatique d'un contrôleur, il génère automatiquement la base du fichier test, au point de vue fonctionnel et unitaire. Il n'est bien sûr créé que le test générique de base mais la démarche est marquée et marquante pour tout nouvel utilisateur.

Il suffit d'ouvrir le répertoire test pour s'en rendre compte :

- fixtures
- functional
- integration
- mocks
- unit

Nous verrons, via les exemples pratiques, que cela nous apporte un bon début de palette de tests. Pour ceux souhaitant entrer plus avant dans le *Test-Driven Development*, des plugins spéciaux tel rSpec[3] permettent de simplifier encore plus la syntaxe pour se rapprocher de l'anglais.

Mais trêve de bavardage, attardons nous sur ces répertoires si bien structurés. Prenons-en un dont le nom nous semble familier.

7.3.2.1 Tests unitaires

Sous le répertoire *unit* vont se retrouver les tests unitaires. Ceux-ci vont nous permettre de malmener le développeur responsable des modèles. En effet, ce sont les modèles (c'est-à-dire les classes même de notre application) qui vont être examinés.

Pour être complet, nous dirons que chaque méthode devra être testée avec les arguments normalement reçus mais aussi les plus inattendus. Un peu de *fuzzing* serait tout à fait profitable.

Les tests se réalisent en créant des instances d'objets, en allouant des variables et en posant des assertions. Par exemple :

- l'objet X est-il valide sur base des validations ?
- l'absence de tel champ lors de la création, provoque-t-elle l'erreur Y ?
- ...

[1] Le principe KISS est une maxime invoquant la simplicité en toute chose. Son extension acronyme traditionnelle est (en anglais) « Keep it Simple, Stupid » ou « Keep It Sweet & Simple » ou encore « Keep It Short & Simple ». http ://en.wikipedia.org/wiki/KISS_principle

[2] Le principe YAGNI quant à lui est de suggérer aux programmeurs de ne pas ajouter une fonctionalité si elle n'est pas nécessaire. Son extension acronyme traditionnelle est (en anglais) « You Ain't Gonna Need It » http ://en.wikipedia.org/wiki/YAGNI

[3] http ://rspec.info/

7.3.2.2 Tests fonctionnels

Changeons maintenant de répertoire : voici les tests fonctionnels à l'honneur via le répertoire *functional*. Nous montons d'un cran dans notre modèle MVC pour nous occuper des contrôleurs. Ici, chaque action des contrôleurs sera passée au crible.

Nous pourrons donc tester sur base des paramètres ou d'autres données reçues (voire même de réponses fictives d'autres contrôleurs) et vérifier si, par exemple :

1. les pages renvoyés sont les bonnes ;
2. les appels au modèle ont été couronnés de succès ;
3. si un compteur a bien été incrémenté ;
4. ...

7.3.2.3 L'intégration globale

Notre préoccupation immédiate est ici une analyse de test global. En d'autres termes, nous testons ici des scénarios d'action, des cas d'utilisation. Nous ferons donc appel à plusieurs contrôleurs pour tester l'ensemble des réactions de l'application pour un scénario donné.

Ce genre de test n'a pas encore été envisagé pour le présent projet.

7.3.2.4 Les fixtures

Un des noms les plus étranges au premier abord dévoile en fait un précieux raccourci. En effet, il ne s'agit ni plus, ni moins que d'instances d'objets précréés. Ainsi, nous créons des instances d'objets-types valides ou non afin de pouvoir les réutiliser lors des tests. Un nom spécifique est donné à chacun (genre de label ou de référence) afin de pouvoir l'appeler facilement. Ces fixtures permettent donc d'alléger le code des tests en les rendant plus lisibles car plus courts.

Avant chaque utilisation de fixtures dans un fichier test, nous devons lier symboliquement le fichier de test au fichier contenant les fixtures. Cela se fait *fixtures :nom_du_fichier_fixtures_sans_l_extension*. Nous pouvons ajouter à la suite un grand nombre de fichiers si nécessaire.

Par exemple : *fixtures :users, :people, :organisms, :categories*

D'un point de vue plus pratique, il est toujours conseillé de nommer les fichiers de manière logique et représentative du contenu par rapport à la classe.

Les fichiers fixtures ont une extension .yml car les données sont formatées en YAML même si le CSV pourrait fonctionner.

7.3.2.5 Les mocks

L'utilité de cette partie est parfois controversée. Ce répertoire sert principalement à créer de faux objets permettant de simuler les réactions d'un serveur distant suite à une requête ou à un autre objet. En effet, lors de la phase de développement, il est préférable de ne pas accéder à l'environnement de production pour différentes raisons.

Cette partie n'a pas été implémentée dans le projet.

7.3.3 Les tests du projet

Analysons ensemble quelques tests.

7.3.3.1 Les tests unitaires

test/unit/person_test.rb

```
1
2  require File.dirname(__FILE__) + '/../test_helper'
3
4  class PersonTest < Test::Unit::TestCase
5    fixtures :people
6  [...]
```

Voici les entêtes des fichiers. Il n'y a pas beaucoup de commentaires à y apporter. La classe de test *PersonTest* hérite directement d'une classe de test mère.

test/unit/person_test.rb

```
1
2  [...]
3   def test_truth
4      assert true
5    end
6
7    def test_validation
8      pers_csa = people(:bernard)
9      assert pers_csa.valid?
10   end
11
12   def test_last_name_invalid
13     pers_csa = people(:miss_last_name)
14     assert !pers_csa.valid?
15     assert pers_csa.errors.invalid?(:last_name)
16   end
17 [...]
```

Voici trois tests typiques : le premier est le test générique, celui de vérité. Celui-ci doit renvoyer *true* vu qu'il n'y a aucune condition.

Le deuxième test est le test classique par excellence : une personne est créée avec toutes les caractéristiques nécessaires. Celle-ci doit être valide. Ce test ratera généralement si une validation supplémentaire est ajoutée à partir d'un nouveau champ.

Le troisième test est le test d'erreur : une personne dont le nom de famille manque est créée. Cette personne ne peut être valide et doit renvoyer une erreur concernant le nom de famille.

test/unit/person_test.rb

```
1
2  [...]
3    def test_function
4      p = Person.new(:first_name => 'John', :last_name => 'Doe', :updated_by => 1)
5      f = Function.new(:title => 'Manager')
6      assert f.save
7      assert p.save
8      assert p.function = f
9      assert_equal p.functions, [f]
10   end
11
12   def test_function
13     p = Person.new(:first_name => 'John', :last_name => 'Doe', :updated_by => 1)
14     f = Function.new(:title => 'Manager')
15     assert p.save
16     assert f.save
17     p.function = f
18     p.reload
19     f.reload
20     assert_equal p.functions, [f]
21     assert_equal p.function, f
22
23     chercheur = Function.new(:title => "Chercheur", :person_id => p.id)
24     assert chercheur.save
25     p.reload
26     assert_equal p.functions.map(&:title), ["Manager", "Chercheur"]
27     assert_equal p.function, f
28
29     assert chercheur.update_attributes(:default_person => 1)
30     p.reload
31     assert_equal p.function, chercheur
32   end
33
34   def test_lonely_function_automatically_set_to_default
35     p = Person.new(:first_name => 'Didier', :last_name => 'Bellens', :updated_by => 1)
36     assert p.save
37     f = Function.new(:person => p, :title => 'CEO')
38     assert f.save
39     p.reload
40     f.reload
41     assert_not_nil Function.find_by_title("CEO")
42     assert f.default_person
43     assert_not_nil p.function
44   end
45 [...]
```

Les trois derniers tests sont en rapport avec une fonction comme vous pouvez le remarquer.

Une personne et une fonction sont créées et chacune doit être valide. Il doit être possible de retrouver la fonction via la personne (1er test).

Si nous créons une deuxième fonction, les deux doivent être accessibles et la première doit toujours être celle par défaut vu que le changement de priorité n'a pas été forcé. (2ème test).

Si une fonction est la seule existante pour une personne, celle-ci doit être la fonction par défaut de cette personne (3e test).

Il est évident que le nombre de tests unitaires peut être multiple pour une même classe. Néanmoins, les fonctions et les liens les plus susceptibles de poser problème ont été envisagés.

Les tests exposés ici ne sont pas limitatifs et leur nombre peut augmenter facilement. Tout est une question de temps disponible ou du temps qu'on s'accorde pour un nombre de tests suffisamment représentatifs.

7.3.3.2 Les tests fonctionnels

test/unit/person_test.rb

```
1  require File.dirname(__FILE__) + '/../test_helper'
2  require 'people_controller'
3
4  # Re-raise errors caught by the controller.
5  class PeopleController; def rescue_action(e) raise e end; end
6
7  class PeopleControllerTest < Test::Unit::TestCase
8    fixtures :people, :users, :categories
9
10   def setup
11     @controller = PeopleController.new
12     @request    = ActionController::TestRequest.new
13     @response   = ActionController::TestResponse.new
14     @first_id = people(:first).id
15     @request.session[:user_id] = users(:first).id
16   end
17 [...]
```

Il y a peu de commentaires à faire concernant les en-têtes. Le *setup* permet de définir des variables accessibles durant l'ensemble des tests pour cette classe. Notons la ligne *@request.session[:user_id] = users(:first).id* permettant l'identification. Sans cette ligne, toutes nos requêtes de pages seraient redirigées vers la page d'identification, ce qui n'aurait aucun intérêt pour la plupart des tests. Ceci n'est bien sûr valable que pour ce projet car nous avons obligé les utilisateurs à s'authentifier avant toute action.

test/unit/person_test.rb

```
1  [...]
2    def test_index
3      get :index
4      assert_response :success
5      assert_template 'list'
6    end
7
8    def test_list
9      get :list
10     assert_response :success
11     assert_template 'list'
12     assert_not_nil assigns(:people)
13   end
14 [...]
```

Ci-dessus, deux tests assez simples d'accessibilité des pages.

test/unit/person_test.rb

```
1  [...]
2    def test_update
3      post :update, :id => @first_id, "controller"=>"people",
4          "item"=>{"organism_id"=>["1", "", "", "", "", ""],
5          "title"=>["Administrateur", "", "", "", "", ""]},
6          "person"=>{"gender"=>"M", "description"=>"Expert juridique",
7          "first_name"=>"Joseph", "last_name"=>"Test"}
8      assert_response :redirect
9      assert_redirected_to :action => 'show', :id => @first_id
10   end
11
12   def test_destroy
13     assert_nothing_raised { Person.find(@first_id) }
14
15     post :destroy, :id => @first_id
16     assert_response :redirect
17     assert_redirected_to :action => 'list'
18     assert_raise(ActiveRecord::RecordNotFound) {
19       Person.find(@first_id)     }
20   end
21 [...]
```

Voici les tests d'actions un peu plus complexes vu l'enchevêtrement de la vue de création. Le deuxième test vérifie également la suppression effective de la personne.

Mentionnons également les tests pour les redirections. Si l'action se déroule bien mais que la page affichée à la suite n'est pas la bonne, le test échoue. D'ailleurs, c'est ainsi que je me suis rendu compte d'avoir omis de m'authentifier pour ces tests.

test/unit/person_test.rb

```
1  [...]
2    def test_delete_category
3        get :show, :id => @first_id
4        person = Person.find(:first)
5        num_category = person.categories.size
6        person.categories << categories(:second)
7        assert_equal person.categories.size, num_category + 1
8        person.categories.delete(categories(:second))
9        assert_equal person.categories.size, num_category
10       assert_response :success
11       assert_template 'show'
12   end
13 [...]
```

Test sur la suppression d'une catégorie : nous pourrions ainsi coupler test d'ajout et test de suppression mais il est préférable de dédoubler le test afin de pouvoir, en cas d'échec, circonscrire directement le problème à la simple vue du nom du test ayant échoué.

7.3.3.3 Les fixtures

Chargeons le fichier des personnes (*test/fixtures/people.yml*).

test/fixtures/people.yml

```
1
2  first:
3    id: 5
4    last_name: Mulder
5    first_name: Fox
6    updated_by: 1
7
8  bernard:
9    id: 1
10   last_name: Dubuisson
11   first_name: Bernard
12   updated_by: 1
13
14 delphine:
15   id: 2
16   last_name: Degreef
17   first_name: Delphine
18   updated_by: 1
19
20 miss_last_name:
21   id: 3
22   first_name: John
23   updated_by: 1
24
25 miss_first_name:
26   id: 4
27   last_name: Doe
28   updated_by: 1
```

7.4 Sécurité

Bien sûr, l'application développée pour cette épreuve intégrée ne sera pas disponible via Internet. Néanmoins, est-ce une raison valable pour ne pas se soucier de la sécurité ?

Outre le caractère formatif, la sécurité est un des thèmes principaux concernant le développement d'applications informatiques de nos jours.

Il me semble néanmoins nécessaire de contraster ce point.

En effet, il y a un équilibre à trouver entre sécurité et commodité. Un ordinateur sans mot de passe est certes pratique mais pas sécurisé. Un ordinateur avec une authentification toutes les 5 minutes grâce à une analyse ADN n'est pas des plus pratiques mais est très sécurisé.

Nous considèrerons donc d'avantage la sécurisation intrinsèque à l'application vis-à-vis de l'extérieur d'un point de vue global.

7.4.1 Authentification

Pour le projet, la mise en place d'une authentification par mot de passe a été privilégiée. Celle-ci repose sur le stockage du hachage du mot de passe.

app/models/user.rb

```
1  [...]
2    def password=(pwd)
3      @password = pwd
4      create_new_salt
5      self.hashed_password = User.encrypted_password(self.password, self.salt)
6    end
7
8  [...]
9
10   private
11
12   def self.encrypted_password(password, salt)
13     string_to_hash = password + "wibble" + salt # 'wibble' makes it harder to guess
14     Digest::SHA1.hexdigest(string_to_hash)
15   end
16
17   def create_new_salt
18     self.salt = self.object_id.to_s + rand.to_s
19   end
20 [...]
```

D'autres authentifications sont néanmoins possibles via plugins. Parmis celles-ci, il existe l'authentification via :

- une Access Control List ;
- un serveur d'authentification centralisé ;
- l'API d'authentification de Google (permet de se logger via son login/mot de passe Google) ;
- ...

7.4.2 Injection SQL

L'injection SQL est LE problème numéro 1 des applications Web. Il est donc important de s'en préoccuper. ActiveRecord prendra soin, via ses fonctions (*attributes, save, find*), de maintenir votre application à l'abri si vous respectez la syntaxe spécifique et non les requêtes SQL en dur dans le code Ruby. Si toutefois vous souhaitez en utiliser, faites attention à ne pas mettre de méta-caractères SQL.

7.4.3 Validation ActiveRecord

Les validations via le modèle sont bien plus pratiques qu'écrire directement les contraintes en SQL. La syntaxe Ruby est assez simple à lire et à utiliser. Les *null* sont bien pris en compte via les *:allow_nil*. S'ils sont associés à la validation d'unicité, des combinaisons intéressantes peuvent être trouvées rapidement sans trop de code.

7.4.4 Création d'enregistrements directement depuis les paramètres

Un danger de l'enregistrement direct des données via un

```
User.create(params[:user])
```

est qu'il suffit à une personne mal intentionnée de modifier les paramètres pour ajouter les champs non repris dans le formulaire mais existants pour risquer de se trouver avec une exploitation abusive de l'application par usurpation d'identité. (Exemples de champs « à risques » : champ définissant les rôles ou champ booléen confirmant l'approbation de l'utilisation dans le réseau)

Il est possible, simplement, d'empêcher la création/modification de certains champs pour une personne alpha.

Cela se fait directement dans la classe du modèle par :

```
attr_protected :champ_à_protéger
```

Inversement, nous pouvons donner accès en modification à des champs spécifiques.

```
attr_accessible :champ_accessible
```

7.4.5 Méthodes privées ou protégées

Il s'agit ici d'une remarque générale : vu qu'une méthode d'un contrôleur qui n'est pas mise en *private* ou *protected* devient une action de ce même contrôleur, une grande attention doit être maintenue sur ce point.

7.4.6 Affichage d'une donnée

Le risque est d'afficher des données auxquelles l'utilisateur n'a pas accès (commandes d'un autre utilisateur, ...).

Il est toujours préférable d'essayer au maximum d'utiliser un élément contrôlable.

Plutôt que d'affecter une variable via une recherche sur la classe (@order = Order.find(order_id)), restreignons notre requête de la façon suivante :

```
@order = @user.orders.find(order_id)
```

Le fait de restreindre permet d'éviter de mauvaises surprises.

7.4.7 Minimiser les attaques de sessions

Rails réduit déjà celles-ci en ne les utilisant qu'avec parcimonie. Néanmoins, réduire la durée de validité de la session permet de perfectionner cette protection.

app/controllers/application.rb

```
1  [...]
2  before_filter :session_expiry
3
4    #Timeout after inactivity of one hour
5    MAX_SESSION_PERIOD = 3600
6    private
7
8    def session_expiry
9      reset_session if session[:expiry_time] and session[:expity_time] < Time.now
10     session[:expiry_time] = MAX_SESSION_PERIOD.seconds.from_now
11     return true
12   end
13 [...]
```

Le filtrage sur les adresses IP est également possible mais cause d'autres soucis.

7.4.8 Attaque XSS (Cross Site Scripting)

Il est question, ici encore, d'injection ou de vol de cookies, ... via le Javascript, entre autres.

Afin d'éviter de telles attaques, il est important de refuser les méta-caractères HTML. Ces derniers peuvent venir d'un input classique par POST ou GET (depuis l'URL, par exemple), de paramètres stockés dans la base de données ou de code XML-RPC type Javascript. Cela ne se limite donc pas qu'au seul Javascript.

Rails possède plusieurs méthodes incluses de base pour faire ce travail dont *escape_html()*, *h()*, *url_encode()*, *sanitize()*, ...

Cette technique a été très utilisée dans les vues du projet.

7.4.9 Autres recommandations

- Utilisation de routines évaluant la difficulté des mots de passe ;
- Transmission d'informations de manière sécurisée (SSL, ...) ;
- Vérification des fichiers en upload ou download ;
- Sécurisation de l'accès à la base de données (SSH, services inutiles sur les machines, ...) ;
- ...

Chapitre 8

Analyse de l'état d'avancement du projet

8.1 Orientation utilisateurs

D'une part, suite à la méthode agile et, d'autre part, vu qu'il s'agit d'une bonne pratique, une importance a été accordée au feedback du client et des utilisateurs finaux.

Malheureusement, il n'a pas été possible de définir un rendez-vous dans les délais de remise du présent travail afin de réaliser un première présentation du programme aux secrétaires. En effet, nous avons eu rarement l'occasion de travailler directement avec elles. Néanmoins, il est prévu que cette rencontre ait lieu avant la défense orale.

Elle aura de nombreux avantages :

- avoir une vraie réaction concernant les champs utilisés, inutiles ou à ajouter ;
- avoir une appréciation des méthodes de saisie mises en place (auto-complétion, lien direct lors la création d'une personne de contact entre l'organisme et la personne, ...) ;
- recevoir les demandes sur les fonctionnalités susceptibles de faciliter le travail quotidien ;
- permettre de prendre des exemples réels en direct avec la personne en charge ;
- ...

Une attention toute particulière fut apportée à la compatibilité entre navigateurs. L'application étant destinée à être utilisée sous 3 systèmes d'exploitation et avec au moins 2 navigateurs. Le site a été testé avec IE4Linux[1] afin que les bugs d'affichage soient minimisés. Le code CSS a également été adapté.

Le côté itératif de la méthode Agile permet en effet d'être très réactif à ce genre de changement.

Le côté analyse UML ou autre a été mis de côté par les développeurs afin de se concentrer plus sur les itérations et leur fonctionnement que sur la structure générale de l'application. Le code évolue donc en permanence via une factorisation, via un objectif d'optimisation des performances et via une maximisation de la sécurité, tout en n'omettant nullement une certaine convivialité du programme devant fonctionner sur des ordinateurs de tous types, ce qui n'est pas un grand souci pour les applications Web. La réactivité de l'application est l'un des points les plus importants.

Un procédé de gestion des conflits transactionnels de la base de données a été mis en place. Ici encore, une grande simplicité a été constatée.

[1] http ://www.tatanka.com.br

8.2 État actuel

Voici une liste des différentes fonctionnalités disponibles :

- CRUD[2] de personnes et d'organismes ;
- Gestion des fonctions sans trace de l'historique ;
- CRUD d'adresses postales, adresses mails liés à des personnes et des organismes ;
- Gestion des catégories via un tagging en auto-complétion ;
- Possibilité de traitement de l'information en exportation CSV/Microsoft Excel ;
- Possibilité de traitements groupés via les paniers ;
- Identification par mot de passe suffisamment sécurisé pour le type d'usage du client ;
- Ergonomie du programme de type CSS ;
- ...

Solution aux problèmes initiaux ? :

√ La nouvelle solution centralisée permettra d'exploiter les données des contacts à des fins diverses.

√ La nouvelle solution conservera l'intégrité et la cohérence des données.

√ La mise en place dans un Intranet permettra à tous d'y accéder de manière sécurisée.

√ L'unicité au point de vue du lieu de stockage de l'information.

× Mise en place d'un système d'autorisation.

Le client s'est montré très satisfait de l'état d'avancement du projet. Il souhaite prolonger le travail afin de finaliser les points encore à résoudre. Le mois de juin servira, par ailleurs, à améliorer l'application.

8.3 Points à améliorer et à ajouter

Les points à ajouter sont à aller chercher dans les demandes initiales.

Comme évoqué à de nombreuses reprises dans ce travail, l'évolution des fonctionnalités s'est faite progressivement. Il a été tenu compte des points les plus importants à réaliser.

Ces points non implémentés représentent donc les choix du développeur et du client.

Outre l'implémentation de permissions via des rôles définis, nous pouvons retenir trois types de fonctionnalités manquantes actuellement :

– les historiques ;
– les traitements ;
– les compléments de données.

[2] Composants permettant la gestion simple d'une collection d'éléments. http ://fr.wikipedia.org/wiki/CRUD

8.3.1 Les historiques

Les historiques sont ici la mémoire de l'application. C'est suivre la chronologie d'une personne ou d'une fonction. Ce point avait été retiré lors du développement initial pour des raisons pratiques et de simplicité. En simplifiant le schéma, nous rendions sa mise en place et sa résolution plus accessible pour les premières étapes du développement. Prendre un seul problème à la fois nous semble avoir été une bonne pratique.

8.3.2 Les traitements

Cette partie est bien sûr l'une des dernières à ajouter mais également celle qui apporte le maximum de productivité. Il s'agit ici de réduire le travail humain par un encodage et une sélection directement via l'appplication, en évitant des pertes de temps entre 3 fichiers Excel n'étant plus à jour, plus un nombre incalculables de copier/coller.

Parmis ces traitements, nous pouvons retrouver l'envoi de mail, la génération de mailing, ...

8.3.3 Les compléments de données

Cette partie sera la prochaine à être implémentée. En effet, c'est par la rencontre avec les secrétaires que nous verrons si elles découvrent des cas non prévus. Nous ne nous faisons pas d'illusion, il y en aura. Cela peut aller de l'adresse postale d'une fonction jusqu'à des données utiles non présentes actuellement dans la description d'une personne ou d'un organisme.

8.4 Difficultés rencontrées

L'un des grands problèmes fut de gérer les agendas réciproques. En effet, le client et moi-même avions pas mal d'autres obligations professionnelles occupant une part importante de notre temps. Vu que celles-ci ont occupé de larges plages de temps à des moments différents pour chacun, les contacts se sont faits parfois un peu rares.

Cela eut principalement des conséquences sur l'analyse initiale, sur le débuggage de certaines parties, sur les explications pratiques au niveau des attentes en termes de fonctionnalités et, bien entendu, sur la programmation en elle-même.

Ruby on Rails, même s'il s'agit d'un framework extrêmement intéressant, n'est pas enseigné à l'École de Commerce et d'Informatique. Une étude personnelle, un apprentissage assidu a dû être mis sur pied afin d'être productif. En effet, la programmation doit être répétée fréquemment afin que la courbe d'apprentissage puisse tendre vers le haut. Cependant, vu que ma profession n'est pas vraiment axée actuellement sur l'informatique, la mémorisation de l'information n'a pas pu être aussi rapide que je l'aurais souhaité.

Ma personnalité a également beaucoup joué. Ayant un style social[3] axé sur l'analyse, je décortique les situations, crée des liens entre les différents éléments et structure mon raisonnement avec diverses sources (revues spécialisées, rapports, enquêtes, tests, benchmarking). Je ne prends jamais une décision sans m'être convenablement renseigné au préalable.

Je privilégie la qualité plutôt que de risquer de passer trop de temps à faire des erreurs.

J'ai donc tendance à me rassurer, à me sécuriser, à me féliciter du travail accompli. De fait, les situations m'obligeant à agir rapidement m'occasionnent un stress conséquent.

Dans un tel environnement en constante mutation, il n'est pas simple de rattraper le temps « perdu » par manque d'informations. Cependant, la méthode agile m'a permis de me concentrer sur de petits objectifs l'un après l'autre. Ce fut donc très positif.

[3] http ://en.wikipedia.org/wiki/Social_style

Chapitre 9

Conclusions

Je me suis lancé dans le langage Ruby par curiosité, il me semblait intéressant, attirant. J'ai eu envie de l'utiliser car il pouvait me permettre d'exprimer ce que je souhaitais et ce de la manière voulue. En effet, Ruby utilise une syntaxe aisément compréhensible et est entièrement orienté objet. Par rapport à d'autres langages comme le C ou le C++, il a l'avantage d'être un langage de haut niveau.

Monsieur Dubuisson, tout autant que moi, trouvait en Ruby et en Ruby on Rails un potentiel formidable à exploiter. Deux personnes vers un même point. J'avais mon client.

L'apprentissage plus poussé de Ruby et Ruby on Rails ne fut pas un long fleuve tranquille. Il y eut des déceptions et la crainte de ne pas y arriver. Heureusement, le client a été là pour écouter, répondre à mes questions et me conseiller. Il fut important dans la réalisation car il avait l'expérience. Il avait déjà trébuché et trouvé des solutions.

Cette collaboration m'a clairement enrichi et j'en ressors grandi. J'ai pu accomplir un projet passionnant qui peut encore évoluer. Il reste important pour moi de montrer ce projet aux utilisateurs et de le peaufiner en fonction de leurs remarques. Sans mon intervention, il me paraît évident que le programme aurait quand même été créé en interne, ou via des consultants. Cependant, le résultat de mon travail sera utilisé professionnellement, ce qui est une grande source de satisfaction et d'accomplissement.

Concernant Ruby, je ne suis encore loin de tout connaître mais j'en sais déjà suffisamment pour avoir envie de continuer à l'utiliser. J'ai acquis, selon moi, de bonnes pratiques qui me seront profitables, ce dont je suis ravi.

Au delà de l'épreuve intégrée, le graduat m'a permis de combler un manque. En effet, l'informatique a toujours été une passion pour moi. J'avais même envisagé sérieusement de suivre une licence à l'Université. Pour diverses raisons, je ne l'ai pas fait, mais j'ai toujours fait de mon mieux pour acquérir des connsaissances durant mes temps libres. Après mes études, j'ai souhaité approfondir ces connaissances de manière plus structurée. Ce sont des moments inoubliables que j'ai passé durant ces années à l'E.C.I. De plus, grâce à ce graduat, j'ai été en mesure de réorienter ma carrière professionnelle.

J'ai appris, j'ai aidé et j'en suis sorti grandi ... merci.

Chapitre 10

Bibliographie

10.1 Sites Internet

- Ruby on Rails Security Guide, Nakul Aggarwal et Ritesh Arora, 2007, [Consulté pour la dernière fois le 08 mai 2008]. Disponible sur : http ://www.quarkruby.com/2007/9/20/ruby-on-rails-security-guide

- Ruby on Rails Security Project, Ruby on Rails Security Project, 2007 [Consulté pour la dernière fois le 08 mai 2008]. Disponible sur : http ://www.rorsecurity.info/

- PeepCode Screencasts, Topfunky Corporation, 2006-2007 [Consulté le 10 décembre 2007], Disponible sur : http ://peepcode.com/

- Gestion des Presenters avec Rspec, Nicolas Mérouze, 2007 [Consulté le 15 décembre 2007]. Disponible sur http ://blog.boldr.fr/2007/9/20/gestion-des-presenters-avec-rspec

- Ruby on Rails, Ruby on Rails Project, 2008, [Consulté pour la dernière fois le 08 mai 2008]. Disponible sur http ://www.rubyonrails.org/

10.2 Livres consultés

- Bersini Hugues et Wellesz Ivan, *L'orienté objet (2e édition)*, Eyrolles, 2004, 550 pages

- Black David, *Ruby for Rails : Ruby Techniques for Rails Developers*, Manning Publications, 2006, 532 pages

- Bradburne Alan, *Practical Rails Social Networking Sites*, Apress, 2007, 421 pages

- Carlson Lucas, *Ruby Cookbook*, O'Reilly Media Inc., 2006, 906 pages

- Cooper Peter, *Beginning Ruby (From Novice to Professional)*, Apress, 2007, 664 pages

- Fernandez Obie,*The Rails Way*, Addison-Wesley Professional, 2007, 912 pages

- Fitzgerald Michael, *Learning Ruby*, O'Reilly Media Inc., 2007, 275 pages

- Flanagan David et Matsumoto Yukihiro, *The Ruby Programming Language*, O'Reilly Media Inc., 2008, 444 pages

- Fowler Chad, *Rails Recipes*, Pragmatic Bookshelf, 2006, 344 pages

- Fulton Hal, *The Ruby Way - Second Edition*,Addison-Wesley Professional, 2008, 888 pages

- Hardy Jeffrey Allan, Carneiro Cloves Jr. et Catlin Hampton, *Beginning Rails From Novice to Professional*, Apress, 2007, 361 pages

- Hartl Michael et Prochazka Aurelius, *Railsspace : Building a Social Networking Website With Ruby on Rails*, Addison-Wesley Professional, 2007, 537 pages

- Hibbs Curt et Tate Bruce A., *Ruby on Rails : Up and Running*, O'Reilly, 2006, 182 pages

- Hunt Andy et Subramaniam Venkat, *Practices Of An Agile Developer : Working In The Real World* , Pragmatic Bookshelf, 2006, 176 pages

- Lenz Patrick, *Build Your Own Ruby on Rails Web Applications*, SitePoint, 2007, 447 pages

- Marshall Kevin, *Web Services on Rails*, O'Reilly, 2006, 32 pages

- Mason Mike, *Pragmatic Version Control : Using Subversion (The Pragmatic Starter Kit Series)(2nd Edition)*, Pragmatic Bookshelf, 2006, 256 pages

- Orsini Rob, *Rails Cookbook*, O'Reilly Media Inc., 2007, 534 pages

- Ortolo Tanguy et Legrand Jeanne, *LaTeX à 200 %*, O'Reilly, 2007, 104 pages

- Richardson Leonard et Ruby Sam, *RESTful Web Services*, O'Reilly Media Inc., 2007, 446 pages

- Sarrion Eric, *Pratique de Ruby on Rails*, O'Reilly, 2006, 579 pages

- Thomas Dave, Fowler Chad et, Hunt Andy, *Programming Ruby : The Pragmatic Programmers' Guide, Second Edition*, Pragmatic Bookshelf, 2004, 864 pages

- Thomas Dave, Hansson David, Breedt Leon, Clark Mike, Davidson James Duncan, Gehtland Justin et Schwarz Andreas, *Agile Web Development with Rails, 2nd Edition*, Pragmatic Bookshelf, 2006, 720 pages

- Williams Justin , *Rails Solutions : Ruby on Rails Made Easy*, friends of ED, 20007, 288 pages

Annexe : Comment installer Ruby on Rails pour le projet

Sous Microsoft Windows

Vous pouvez copier les répertoires de l'application sur votre disque dur.

Environnement Ruby

Rendez-vous sur la page http ://rubyinstaller.rubyforge.org/wiki/wiki.pl et téléchargez la dernière version one-click, self-contained Windows installer.

Les éléments utiles sont l'installation de base, RubyGems et le clavier français (cfr Figure 10.1).

Ruby on Rails

Ouvrez une fenêtre d'invite de commande et saisissez la commande

```
gem install rails -v 1.2.6 --include-dependencies
```

En effet, nous ne prenons pas la dernière version de Rails.

RubyGems va également installer l'une ou l'autre bibliothèque dont Rails dépend.

Base de données

J'ai personnellement choisi de travailler avec MySQL.

Les fichiers de configuration sur le CD sont donc paramétrés en conséquece.

Je pense ne pas devoir détailler plus l'installation de la base de données.

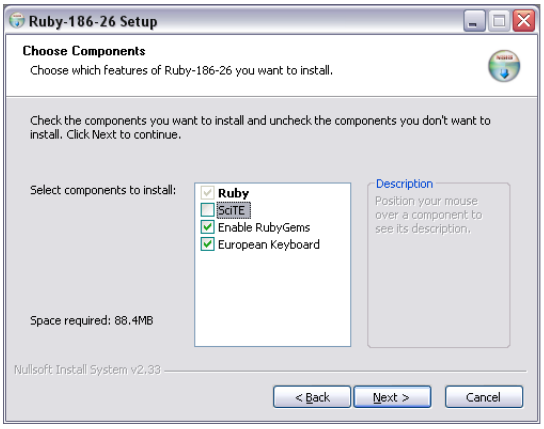


Fig. 10.1 – Installation Ruby sous Microsoft Windows

Il faudra juste remplacer la ligne :

```
socket: /var/run/mysqld/mysqld.sock
```

par :

```
socket: /tmp/mysql.sock
```

dans *config/database.yml* mais aussi créer les bases de données *contacts_development* et *contacts_test* (si on désire tester).

Il sera également nécessaire de modifier le fichier selon votre utilisateur pour la base de données ainsi que son mot de passe.

La création se fait très facilement en faisant un *rake db :migration* dans le répertoire de l'application.

Plugins

Pour l'exportation en CSV, il suffit d'ouvrir une fénêtre d'invite de commande et de saisir :

```
gem install fastercsv
```

Pour l'exportation en Excel :

```
ruby script/plugin install http://svn.napcsweb.com/public/excel
```

dans le répertoire de travail.

Mais ensuite

Vous pouvez ouvrir une fenêtre d'invite de commande et saisir dans le répertoire de travail :

```
ruby script/server
```

Le serveur web intégré Webrick se lancera. Vous n'avez plus qu'à prendre votre navigateur Web préféré et aller à l'adresse : *http ://localhost :3000*.

Il existe un utilisateur *admin* ayant un mot de passe : *admin*

Sous GNU/Linux Ubuntu

Je vous propose ici une installation standard mais complète sous GNU/Linux Ubuntu. Pour les autres distributions, vous pourrez trouver assez facilement les équivalences.

Vous pouvez copier les répertoires de l'application sur votre disque dur.

Environnement Ruby

Dans un terminal

```
~$ sudo apt-get install ruby rdoc irb libyaml-ruby libzlib-ruby ri libopenssl-ruby ruby1.8-dev build-essential
```

RubyGems

```
~$ wget http://rubyforge.org/frs/download.php/34638/rubygems-1.1.1.tgz
~$ tar -xvzf rubygems-1.1.1.tgz
~$ rm rubygems-1.1.1.tgz
~$ cd rubygems-1.1.1
~$ sudo ruby setup.rb
~$ sudo rm -rf rubygems-1.1.1/
```

Ruby on Rails

Ouvrez un terminal et saisissez la commande

```
gem install rails -v 1.2.6 --include-dependencies
```

En effet, nous ne prenons pas la dernière version de Rails.

RubyGems va également installer l'une ou l'autre bibliothèque dont Rails dépend.

Base de données

J'ai personnellement choisi de travailler avec MySQL.

Les fichiers de configuration sur le CD sont donc paramétrés en conséquence.

```
~$ sudo apt-get install libmysql-ruby mysql-server
~$ sudo apt-get install libmysqlclient15-dev
~$ rm rubygems-1.1.1.tgz
~$ sudo gem install mysql
```

Il faudra juste créer les bases de données *contacts_development* et *contacts_test* (si on désire tester).

Il sera également nécessaire de modifier le fichier selon votre utilisateur pour la base de données, ainsi que son mot de passe.

La création se fait très facilement en faisant un *rake db :migration* dans le répertoire de l'application.

Plugins

Pour l'exportation en CSV, il suffit d'ouvrir un terminal et de saisir :

```
gem install fastercsv
```

Pour l'exportation en Excel :

```
ruby script/plugin install http://svn.napcsweb.com/public/excel
```

dans le répertoire de travail.

Mais ensuite

Vous pouvez ouvrir une fenêtre d'invite de commande et saisir dans le répertoire de travail :

```
ruby script/server
```

Le serveur web intégré Webrick se lancera. Vous n'avez plus qu'à prendre votre navigateur Web préféré et aller à l'adresse : *http ://127.0.0.1 :3000*.

Il existe un utilisateur *admin* ayant un mot de passe : *admin*