Jallal Boudabza
Yacine Maghezzi

Master Informatique 2eme année

Département de Mathématique et Informatique

Systèmes d'Informations Orientés Objets
Rapport Préliminaire de Projet

Délai de remise : 23 novembre 2013

Encadrant : Kokou Yetongnon

Table des matières

1 Introduction

De nos jours, avec l'explosion des systèmes connectés et la numérisation de l'information, la base de données est un mal nécessaire afin de pouvoir ranger de manière structurée les informations. Dans ce rapport, notre ambition est de pouvoir expliquer nos choix, nécessaire à la réussite du projet. Pour rappel, nous devons créer un système d'informations orienté objet pour la gestion de *l'université libre de Quetigny*, fictive bien évidemment. Il est a noter que les choix que nous expliquerons tout au long de ce rapport, ne sont pas définitifs, mais seront suivis dans la grande majorité. Au cours du développement, il est possible que certains aspects ne soient pas revisités et seront judicieusement changés.

2 Pré-Requis

FIGURE 2.1 – Logo de notre application

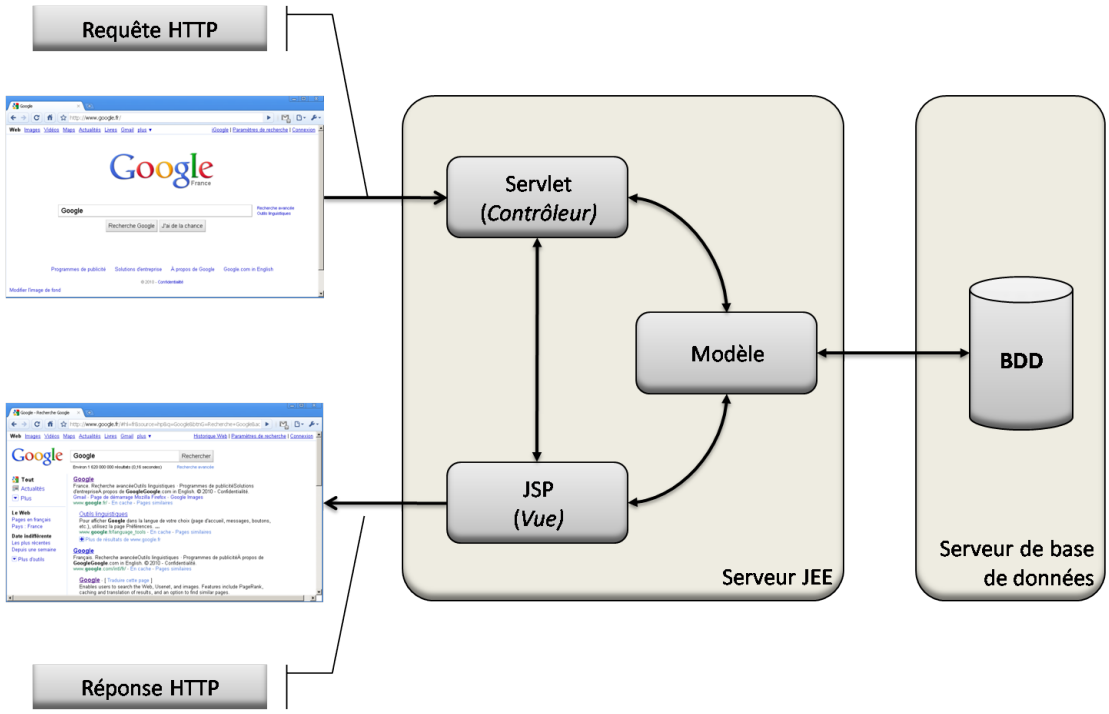
Dans ce projet, nous allons implémenter un système d'informations complet, qui comprendra plusieurs modules de développement. L'idée est de modéliser un système complet, pouvant être déployé sur une structure donnée. L'application doit être fonctionnelle et permettre de gérer cette université dont nous avons crée le logo ci dessus. Nous accorderons une grande importance au choix de l'implémentation du système en essayant de se rapprocher le plus possible d'une application délivrée par par des professionnels. Le choix de la gestion de la persistance des données sera expliqué dans les parties ci dessous.

2.1 Langage de programmation

Le langage que nous utiliserons est java. En effet ce langage nous permettra une connexion à différentes bases de données de manière simplifiée. La richesse de la documentation, et les interfaces entre les différentes SGBD nous permettent d'obtenir une certaine flexibilité. Java nous fourni un panel de drivers d'interfacage entre l'application Web, et le SGBD choisi. On citera par exemple *JDBC*, utile pour les bases de données traditionnelles ou encore Oracle, mais bien d'autres encore qui nous permettront d'interroger une éventuelle base NoSQL.

3 Fonctionnement général

Afin que notre application soit la plus proche d'une application réelle, nous avons opté pour une structure classique, utilisée dans la plupart des applications de ce type.



3.1 Modèle MVC

Le patron modèle-vue-contrôleur, est un modèle destiné à répondre aux besoins des applications interactives en séparant les problématiques liées aux différents composants au sein de leur architectures respectives. Ce paradigme regroupe les fonctions nécessaires en trois catégories :

- un modèle (modèle de données)
- une vue (présentation, interface utilisateur)
- un contrôleur (logique de contrôle, gestion des événements, synchronisation)

Ici, notre vue correspondra à l'interface visuelle proposée aux utilisateurs, notre modèle nous permettra de respecter la persistance des données en insérant ou en rappelant des entités de la base de données, et notre contrôleur, permettra de gérer les requêtes des clients sur notre application.

4 Base de données

4.1 Choix du SGBD

MongoDB est un système de gestion de base de données orientée documents, répartissable sur un nombre quelconque d'ordinateurs, efficace pour les requêtes simples, et ne nécessitant pas de schéma prédéfini des données. **MongoDB** fait parti des SGBD dit NoSQL. Cependant le vrai intérêt de MongoDB par rapport aux concurrents, c'est sa gestion de requêtages. On peut requêter de façon très fine grâce à une grande quantité de mots-clé. On peux donc faire des requêtes aussi riches qu'en SQL, mais tout ça sur une base de données orientée documents.

4.2 Diagramme de classes

Voici le diagramme de classe général de l'application. Afin de mieux comprendre le fonctionnement ainsi que l'architecture de notre base de données, il est nécessaire de faire un diagramme de classes. Les relations, classes ainsi que les cardinalités seront en grande partie respectés sauf cas de force majeur.

Les figures 4.1 et 4.2 correspondent donc au diagramme de classes. Le diagramme a été scindé en 2 parties plus ou moins logiques, dont nous allons faire l'explication.

Dans cette première partie du diagramme de classes (Figure 4.1), nous voyons les classes suivantes :

- Personne (est hérité par Personnel et par Etudiant et est associé à Cours)
- Etudiant (hérite de la classe Personne)
- Bureau (est associée à Personnel)
- LabTP (est associée à Bureau)
- Campus (est associé à Personne)
- Personnel (hérite de Personne et est hérité par Administrateur, Technicien, ChargeCours et Tuteur)
- Administrateur (hérite de Personnel)
- Technicien (hérite de Personnel)
- Tuteur (hérite de Personnel)
- ChargeCours (hérite de Personnel, est héritée par EnseignantAssocie et Enseignant- Vacataire et est associée à Cours)
- EnseignantAssocie (hérite de ChargeCours)

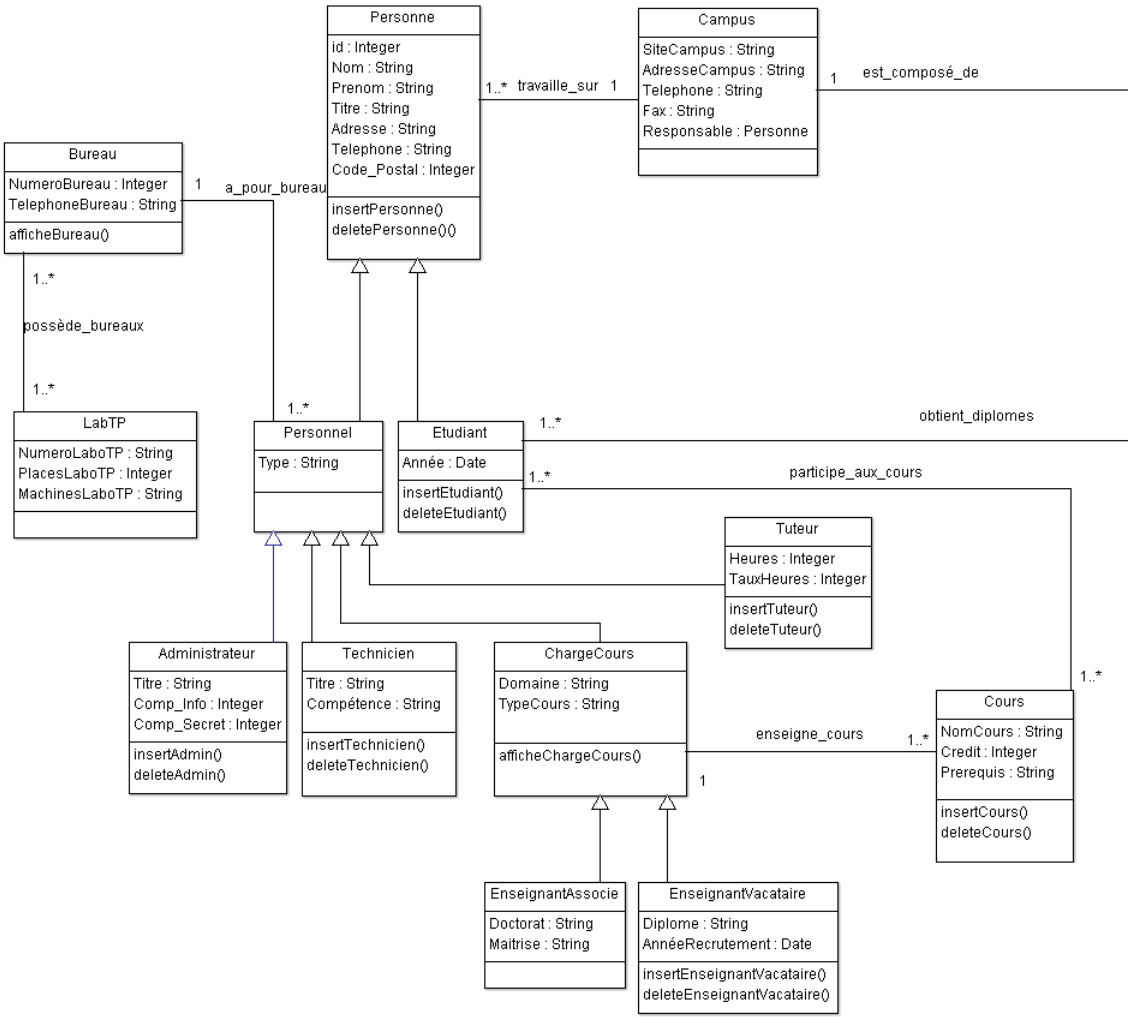


FIGURE 4.1 – Partie gauche du diagramme

– EnseignantVacataire (hérite de ChargeCours)
– Cours (est associé à ChargeCours)

Dans la figure 4.1 nous remarquons qu'il y a 6 classes provenant d'héritage. En effet, Administrateur, Technicien, ChargeCours, Tuteur héritent toutes les 4 de Personnel. Il y a aussi EnseignantVacataire et EnseignantAssocie qui hérite de la classe ChargeCours. Donc le personnel est composé d'administrateurs, de techniciens, de chargés de cours et de tuteurs. Un chargé de cours peut être un enseignant vacataire ou un enseignant associé. Un chargé de cours (enseignant vacataire ou associé) enseigne plusieurs cours, un cours est enseigné par un chargé de cours. Un étudiant participe à des cours, un cours est suivi par plusieurs étudiants. Un étudiant obtient des diplômes, un diplôme peut être obtenu par plusieurs étudiants.

Il existe une association entre Personnel et Bureau car un membre du personnel a un bureau, et un bureau peut appartenir à un ou plusieurs membre du personnel. Une personne fréquente un campus, mais un campus peut être fréquenté par plusieurs personnes.

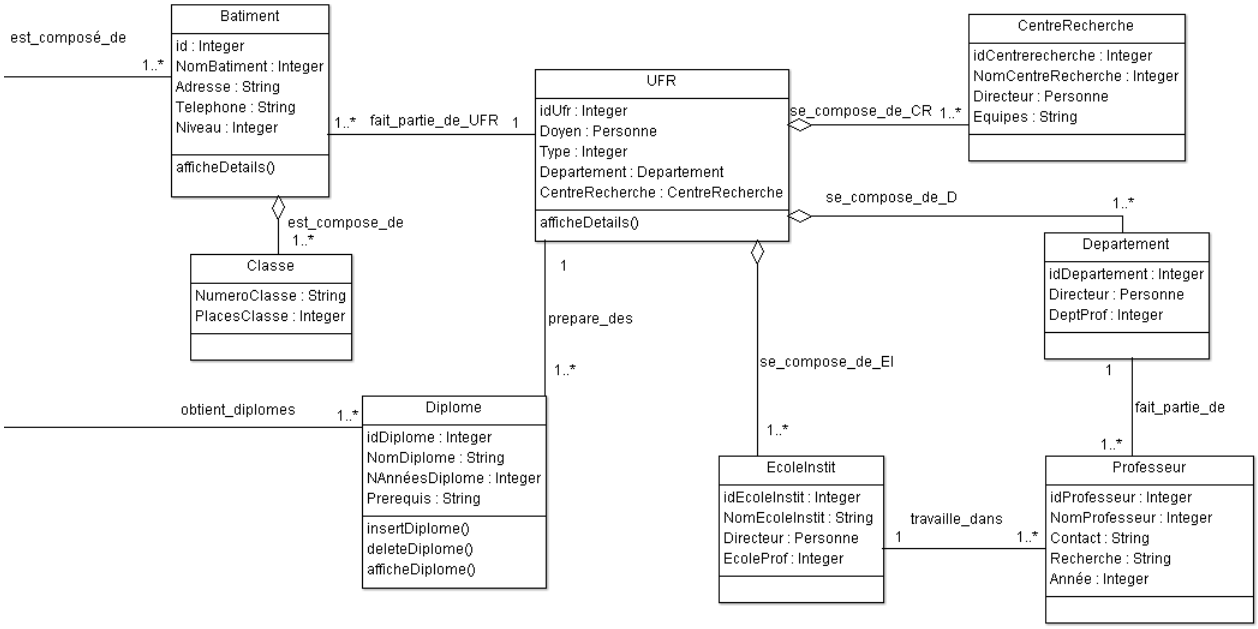


FIGURE 4.2 – Partie droite du diagramme

Dans cette deuxième partie du diagramme de classes (Figure 4.2), nous voyons les classes suivantes :

- Batiment (est composé de Classe)
- Classe (compose un Batiment)
- UFR (est associée à Diplome, Batiment et est composé de EcoleInstitut, CentreRecherche et Departement)
- Diplome (est associé à UFR et Etudiant)
- CentreRecherche (compose l'UFR)
- Departement (compose l'UFR et est associé à Professeur)
- EcoleInstitut (compose l'UFR et est associé à Professeur)
- Professeur (est associé à Departement et à EcoleInstitut)

Un campus est composé de plusieurs bâtiments, un bâtiment faire partie d'un campus. Les bâtiment d'un campus font partie d'une UFR. Un bâtiment est composé de plusieurs classes.

Une UFR permet de préparer des diplômes, un diplôme est obtenu dans une UFR en particulier. Une UFR se compose de centres de recherche, de département et d'école institution. Et un professeur travaille dans une école institution et fait partie d'un département. Un département et une école institution ont chacun plusieurs professeurs.

4.3 Modele ODMG

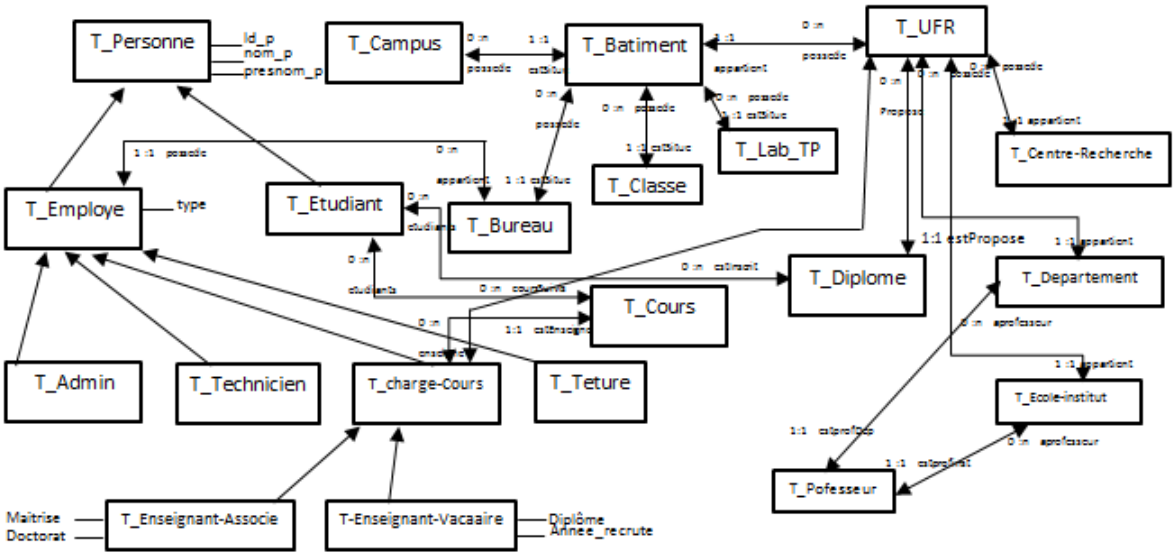


FIGURE 4.3 – Diagramme ODMG

4.4 La syntaxe ODL

Afin de mettre en œuvre la conception de notre base de donnes orientées objets *U-Quetigny* nous allons décrire les différentes classes utilisées.

Tout d'abord, nous avons la classe *Personne* :

```
class Personne
    (extent LesPersonnes key idPersonne)
{
    attribute int id;
    attribute string Nom;
    attribute string Prenom;
    attribute string Titre;
    attribute string Adresse;
    attribute string Telephone;
    attribute string Code_Postal;

    relationship List<Campus> travaille_sur inverse Campus::a_pour_professeur;

    void insertP(idPersonne);
    void deleteP(idPersonne);
}
```

Dans la classe *Personne* nous retrouvons la déclaration des différents attributs tels que *idPersonne*, *Nom-Personne* ou encore *PrenomPersonne*. Tous ces attributs sont décrits dans le diagramme de classes de la figure **??**.

Nous retrouvons également les méthodes suivantes : *insertPersonne* et *deletePersonne*, chacune de ses méthode prend en paramètres l'identifiant de la personne et l'insère ou la supprime de la base de donnes. Nous avons aussi la déclaration des liens entre les classes, par exemple la classe *Personne* est associée à la classe *Campus*, en effet nous pouvons affirmer qu' *une personne travaille sur un campus* et qu'*un campus a pour professeurs*

Ensuite, nous nous intéresserons aux classes *Personnel* et *Etudiant* du diagramme de la figure 4.1 car elles héritent de la classe *Personne*. Or il est intéressant et nécessaire de savoir gérer ce genre de classes.

Classe *Personnel* :

```
class Personnel : Personne
{
    attribute string TypePersonnel;

    relationship List<Bureau> a_pour_bureau inverse Bureau::appartien_a;
}
```

Classe *Etudiant* :

```
class Etudiant : Personne
{
    attribute date Annee;

    relationship Set<Diplome> obtient_diplomes inverse Diplome::est_obtenu_par;
    relationship Set<Cours> suit_cours inverse Cours::est_suivi_par;

    void insertEtudiant();
    void deleteEtudiant();
}
```

La classe *Personnel* possède également des classes filles.
Classe *Administrateur* :

```
class Administrateur : Personnel
{
    attribute string Titre;
    attribute int Comp_Info;
    attribute int comp_secret;

    void insertAdmin();
    void deleteAdmin();
}
```

Classe *Technicien* :

```
class Technicien : Personnel
{
    attribute string Titre;
    attribute string Competence;

    void insertTechnicien();
    void deleteTechnicien();
}
```

Classe *Tuteur* :

```
class Tuteur : Personnel
{
    attribute int Heures;
    attribute int TauxHeures;

    void insertTuteur();
    void deleteTuteur();
}
```

Classe *ChargeCours* :

```
class ChargeCours : Personnel
{
    attribute string Domaine;
    attribute string TypeCours;

    relationship Cours enseigne_cours inverse Cours::est_enseigne_par;

    void afficheChargeCours();
}
```

La classe *ChargeCours* possède également des classes filles.

Classe *EnseignantAssocie* :

```
class EnseignantAssocie : ChargeCours
{
    attribute string Doctorat;
    attribute string Maitrise;
}
```

Classe *EnseignantVacataire* :

```
class EnseignantVacataire : ChargeCours
{
    attribute string Diplome;
    attribute date AnneeRecrutement;

    void insertEnseignantVacataire();
    void deleteEnseignantVacataire();
}
```

Classe *Campus* :

```
class Campus
{
    attribute string SiteCampus;
    attribute string AdresseCampus;
    attribute string Telephone;
    attribute string Fax;
    attribute Personne Responsable;

    relationship Personne est_frequente_par inverse Personne::frequente;
    relationship Batiment est_compose_de inverse Batiment::fait_partie_du_campus;
}
```

Classe *Cours* :

```
class Cours
    (extent LesCours key idCours)
{
    attribute int idCours;
    attribute string NomCours;
    attribute int Credit;
    attribute string Prerequis;

    relationship List<ChargeCours> est_enseigne_par inverse ChargeCours::enseigne_cours;
    relationship Set<Etudiant> est_suivi_par inverse Etudiant::suit_cours;

    void insertCours();
    void deleteCours();
}
```

Classe *Diplome* :

```
class Diplome
    (extent LesDiplomes key idDiplome)
{
    attribute int idDiplome;
    attribute string NomDiplome:
    attribute int NAnneesDiplome;
    attribute string Prerequis;

    relationship Set<Etudiant> est_obtenu_par inverse Etudiant::obtient_diplomes;
    relationship List<UFR> est_prepare_dans inverse UFR::prepare;

    void insertDiplome();
    void deleteDiplome();
    void afficheDiplome();
}
```

Classe *UFR* :

```
class UFR
    (extent LesUFR key idUFR)
{
    attribute int idUFR;
    attribute Personne Doyen;
    attribute int TypeUFR;
    attribute Departement DepartementUFR;
    attribute CentreRecherche CentreRechercheUFR;

    relationship Batiment est_compose_de_batiments inverse Batiment::fait_partie_de_UFR;
    relationship Diplome prepare_diplomes inverse Diplome::est_prepare_dans;

    relationship EcoleInstit se_compose_decole inverse EcoleInstit::ecole_fait_partie_de;
    relationship CentreRecherche se_compose_de_centre inverse CentreRecherche::
        centre_fait_partie_de;
    relationship Departement se_compose_de_dep inverse Departement::dep_fait_partie_de;

}
```

Classe *CentreRecherche* :

```
class CentreRecherche
    (extent LesCentresRecherche key idCentreRecherche)
{
    attribute int idCentreRecherche;
    attribute string NomCentreRecherche;
    attribute Personne DirecteurCentreRecherche;
    attribute string Equipes;

    relationship List<UFR> centre_fait_partie_de inverse UFR::se_compose_de_centre;
}
```

Classe *Departement* :

```
class Departement
    (extent LesDepartement key idDepartement)
{
    attribute int idDepartement,
    attribute Personne DirecteurDepartement;
    attribute int DeptProf;

    relationship List<UFR> dep_fait_partie_de inverse UFR::se_compose_de_dep;
    relationship Professeur dep_a_pour_professeur inverse Professeur::fait_partie_de
}
```

Classe *EcoleInstit* :

```
class EcoleInstit
    (extent LesEcolesInstit key idEcoleInstit)
{
    attribute int idEcoleInstit;
    attribute string NomEcoleInstit;
    attribute Personne DirecteurEcoleInstit;
    attribute int EcoleProf;

    relationship List<UFR> ecole_fait_partie_de inverse UFR::se_compose_decole;
    relationship Professeur ecole_a_pour_professeur inverse Professeur::travaille_dans
}
```

Classe *Professeur* :

```
class Professeur
    (extent LesProfesseurs key idProfesseur)
{
    attribute int idProfesseur;
    attribute string NomProfesseur;
    attribute string Contact;
    attribute string Recherche;
    attribute int Annee;

    relationship List<EcoleInstit> travaille_dans inverse EcoleInstit::
        ecole_a_pour_professeur;
    relationship List<Departement> fait_partie_de_dep inverse Departement::
        dep_a_pour_professeur;
}
```

4.5 Insertion de tuples

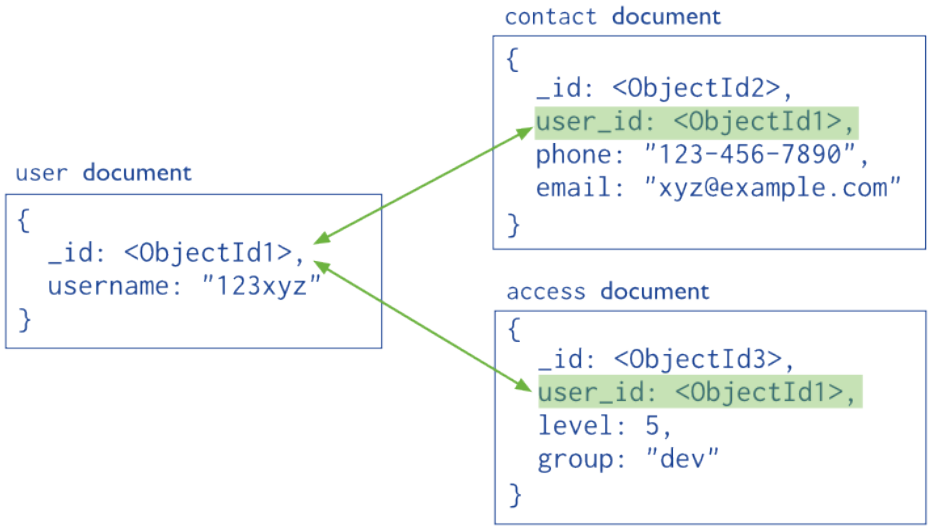


FIGURE 4.4 – Fonctionnement de MongoDB

MongoDB est un SGBD orienté document. Il est donc important d'en comprendre le fonctionnement. Les données sont liées entre elle via des références entre entités, comme expliques dans le schémas précédent.

Voici un exemple d'insertion dans la classe Personne sur **MongoDB** :

```
db.personne.insert( { nom: "Boudabza", prenom: "Jallal", Titre: "Monsieur", Adresse: "Chemin
    de st Apo" ....}
```

4.6 Requêtes OQL

Voici l'exemple de quelques requêtes sur le schémas crée précédemment.

Requête n°1

```
SELECT struct (p.nom, p.prenom)
FROM p in Personne
GROUP BY (Dijonnais: p.codepostal = 21000 ,
Autre : p.codepostal != 21000);
```

Explication : Affichage des de toutes les personnes provenant de Dijon et des alentours.

Requête n°2

```
SELECT c.* FROM etudiant e, cours c WHERE e.id = c.idCours
 AND e.nom = "Maghezzi" AND e.prenom = "Yacine";
```

Explication : Affichage des cours de l'étudiant Maghezzi Yacine.

La requête sur **MongoDB** donnera :

```
etu_temp = db.etu.findOne({nom : « Maghezzi », prenom : « Yacine »})
cours = db.cours.find({idCours : etu_temp.idCours})
```

5 Conclusion

Afin de conclure ce rapport, nous nous sommes rendus compte que l'analyse préliminaire d'un projet, nous permet un gain de temps non négligeable, si les choix sont judicieusement élaborés. La création des différents diagrammes sont essentiels à la réalisation d'une structure réfléchie et cohérente par rapport aux résultats que l'on veut obtenir.

Pour finir, ce rapport est nommé à juste titre préliminaire, les spécifications sont donc amenées à évoluer. La partie pratique une fois terminée, sera décrite dans le rapport final dans lequel nous serons amenés à rediscuter et corriger les choix fait dans ce rapport.