



# MoneyThoring

## RAPPORT FINAL

Daniel Gonzalez Lopez, Guillaume Zaretti,  
François Burgener, Bryan Curchod, Héléna Reymond

A l'intention de M. René Rentsch

FEVRIER 2018

## Table des matières

<b>1</b>	<b>Introduction.....</b>	<b>3</b>
<b>2</b>	<b>Objectifs du projet .....</b>	<b>4</b>
<b>3</b>	<b>Concepts généraux.....</b>	<b>5</b>
<b>4</b>	<b>Fonctionnalités implémentées .....</b>	<b>6</b>
4.1	Compte utilisateur.....	6
4.2	Connexion sécurisée.....	6
4.3	Comptes bancaires .....	6
4.4	Catégories .....	6
4.5	Transactions .....	6
4.6	Dettes.....	7
4.7	Vue globale .....	7
4.8	Budget .....	7
4.9	Budgets partagés .....	7
<b>5</b>	<b>Conception et architecture du projet .....</b>	<b>8</b>
5.1	Technologies utilisées.....	8
5.2	Architecture de la solution .....	8
<b>6</b>	<b>Bases de données (DB) .....</b>	<b>10</b>
6.1	Schéma relationnel.....	10
6.2	PostgreSQL .....	11
6.3	Derby.....	12
6.4	Difficultés rencontrées .....	12
<b>7</b>	<b>Couche d'accès aux données (DAL).....</b>	<b>13</b>
7.1	Projet Maven .....	13
7.2	Mise en place de la DAL.....	18
7.3	Hibernate .....	20
7.4	Difficultés rencontrées .....	20
<b>8</b>	<b>Interface graphique (GUI).....</b>	<b>21</b>
8.1	Fenêtre de connexion/enregistrement d'un compte utilisateur .....	21
8.2	Menu latéral de l'application.....	22
8.3	Fenêtre principale .....	22
8.4	Comptes bancaires .....	23
8.5	Catégories .....	25

---

8.6	Transactions .....	27
8.7	Budgets.....	29
8.8	Dettes.....	33
8.9	Difficultés rencontrées .....	35
<b>9</b>	<b>Logique métier (BLL) .....</b>	<b>37</b>
9.1	Introduction .....	37
9.2	Classes Authentication, KeyGenerator et Mail .....	37
9.3	Classe ClientLogic .....	37
9.4	Les autres classes logiques .....	38
9.5	Mappeurs .....	38
9.6	Difficultés rencontrées .....	39
<b>10</b>	<b>Tests.....</b>	<b>40</b>
10.1	Liste des tests effectués .....	40
10.2	Bugs restants.....	44
<b>11</b>	<b>Conclusion .....</b>	<b>45</b>
11.1	Déroulement du projet.....	45
11.2	Fonctionnement du groupe.....	46
11.3	Avis personnel des membres du groupe .....	48
11.4	Améliorations possibles.....	52
<b>12</b>	<b>Bibliographie / webographie .....</b>	<b>53</b>
12.1	BLL.....	53
12.2	DAL .....	53
12.3	DB.....	53
12.4	GUI .....	54
<b>13</b>	<b>Table des illustrations.....</b>	<b>55</b>
13.1	Architecture.....	55
13.2	Bases de données (DB) .....	55
13.3	Couche d'accès aux données (DAL) .....	55
13.4	Interface graphique (GUI) .....	55
<b>14</b>	<b>Annexes .....</b>	<b>57</b>
14.1	Cahier des charges initial du projet.....	57
14.2	Planification du projet .....	57
14.3	Journaux de travail .....	61

## 1 Introduction

Avec MoneyThoring, nous proposons à l'utilisateur une gestion poussée de leur capital. L'application permet en effet de créer des vues pour ses comptes bancaires, pour lesquels il est possible d'enregistrer des transactions, qu'elles soient de simples dépenses ou des revenus. Nous lui apportons aussi la possibilité de catégoriser ses transactions, de telle sorte qu'il soit plus aisément de voir les transactions d'un même groupe pour l'utilisateur. Ce dernier peut aussi créer des objectifs de budget pour pouvoir suivre ses dépenses dans des domaines précis et ainsi d'avoir une vue la plus informative possible sur l'évolution de son capital. L'utilisateur a également la possibilité d'enregistrer des dettes ou des créances, qui une fois validées seront automatiquement transformées en transaction par le logiciel, ce qui évite une saisie en plus.

Ainsi, MoneyThoring est une application proposant à l'utilisateur de créer un compte en ligne ou uniquement en local. Selon ce choix, certaines fonctionnalités sont accessibles ou non.

Les fonctionnalités concernées sont les budgets partagés et une fonctionnalité liée aux dettes. En effet, MoneyThoring permet à plusieurs utilisateurs du service de se partager un même budget, ce qui permet de suivre les dépenses d'un groupe d'utilisateurs. Par exemple, il serait possible d'utiliser un budget partagé pour gérer et suivre les dépenses lors d'un départ en voyage entre amis ou encore lors d'une préparation de fête, où plusieurs personnes vont participer dans les dépenses. En ce qui concerne les dettes, l'utilisateur a la possibilité de lier une dette ou une créance à un autre utilisateur de l'application. De cette manière, les deux ont la même information et lorsque l'utilisateur confirme le paiement, une transaction est créée chez les deux utilisateurs.

Ces fonctionnalités, concernant d'autres utilisateurs du service, ne peuvent pas être gérées si l'on choisit de partir avec le mode hors ligne, localement. Exception faite de ces dernières, tout est réalisable en local, avec ou sans connexion internet. Cela dit, le mode hors ligne a l'avantage de pouvoir être utilisé même avec une panne de réseau, alors que le mode en ligne dépend totalement de la connexion internet. De plus, ce dernier peut ralentir l'exécution du logiciel, devant faire des requêtes sur une base de données distante, dont les performances peuvent varier.

MoneyThoring propose un design poussé pour que l'information soit la plus claire et précise et que l'utilisateur n'ait pas à se creuser la tête pour comprendre son utilisation. Son design est basé sur les règles de design proposées par Google, le *Material Design*.

## 2 Objectifs du projet

Notre objectif principal était bien évidemment de proposer un produit finit de notre logiciel, permettant ainsi à un utilisateur de gérer son capital de manière simple et visuelle grâce aux différentes fonctionnalités implémentées (transactions, dettes, budgets, ...).

Un autre objectif était d'avoir une bonne base de fonctionnalités implémentées de telle sorte à pouvoir utiliser notre propre application de façon personnelle une fois celle-ci terminée. En effet, nous avons décidé de faire un tel logiciel car il est très utile de pouvoir gérer ses dépenses et les logiciels de la sorte ont toujours une ou plusieurs fonctionnalités manquantes. En implémentant notre propre version, nous pouvons lui ajouter ce que nous estimons utile et nécessaire et la modifier si besoin par la suite, pour qu'elle évolue avec nos besoins.

Comme suggéré dans le paragraphe précédent, un des objectifs du projet était de pouvoir poursuivre l'implémentation et éventuellement étendre son champ d'action, par exemple avec des applications mobiles ou un site web. Bien sûr, avant de penser à l'expansion du service, nous devrions l'améliorer dans notre version de bureau pour qu'il soit impeccable et ainsi éviter de propager des erreurs ou des mauvais concepts.

Un autre objectif de ce projet était de mettre en pratique des connaissances dans un domaine qui plaît à chacun des membres de l'équipe, ainsi que de les perfectionner tout au long du projet. Voir l'ampleur d'un tel travail, l'effort à fournir et les conséquences qui en découlent pour en tirer un maximum d'expérience était un point important du projet.

De même que d'apprendre à utiliser de nouvelles technologies et de pouvoir partager son expérience avec les autres membres du groupe, ayant tous des visions et des compétences différentes.

Enfin, dernier objectif et non des moindres, c'était d'avoir un produit final qui plaise à l'équipe et dont nous serions fiers.

### 3 Concepts généraux

Utilisation du framework Hibernate pour l'accès aux données des bases de données distante et locale et pour la persistance de celles-ci.

Utilisation de PostgreSQL pour la base de données distante, choisi pour sa bonne gestion des logs, qui peuvent devenir utiles lorsque l'application prendra de l'ampleur.

Utilisation de Derby pour la base de données locale, choisi principalement parce que Derby est plus adapté à une application Java que SQLite.

Design de l'application d'après les règles de design proposées par Google, appelées *Material Design (Quantum Paper)*.

Utilisation de JFoenix, librairie JavaFX, prenant en charge les concepts du *Material Design*.

## 4 Fonctionnalités implémentées

### 4.1 Compte utilisateur

Si l'utilisateur ne possède pas de compte à l'ouverture de l'application, il peut s'en créer un. La création du compte se fait à l'aide d'un formulaire qui lui demande son adresse email, son nom d'utilisateur et un mot de passe, qu'il est nécessaire de confirmer une deuxième fois. Une fois le formulaire envoyé, dans le cas du mode en ligne, un email de validation contenant un code d'activation est envoyé à l'adresse email renseignée. Une fois le code saisi dans l'application lors de la première connexion, le compte est activé.

### 4.2 Connexion sécurisée

Pour tout utilisateur possédant un compte, une connexion est exigée au démarrage de l'application. Cette connexion évite qu'un tiers puisse modifier les données sans l'accord de l'utilisateur. Cette demande de connexion est un simple formulaire dans lequel l'utilisateur doit entrer son nom d'utilisateur, ou son adresse email, et son mot de passe. L'application ne nécessite pas de double authentification.

Le mot de passe est haché avec un sel généré lors de l'enregistrement dans la base de données. Il n'est jamais sauvé en clair.

### 4.3 Comptes bancaires

Un utilisateur peut ajouter un ou plusieurs comptes bancaires. Chaque compte possède un nom, un type, le nom de la banque qui le concerne, le montant actuel et s'il faut l'utiliser comme compte par défaut lors des transactions. Toutes ces informations, excepté le solde du compte qui est modifié automatiquement par les transactions, peuvent être modifiées par la suite. Les comptes bancaires peuvent également être supprimés, ce qui ne les supprime pas réellement de la base de données mais les rend juste invisible, pour éviter de perdre les transactions qui y sont liées.

### 4.4 Catégories

Il est possible de créer d'autres catégories que celle proposée par défaut. Une catégorie est définie par un nom et une couleur. Toutes les catégories peuvent être modifiées, exception faite du nom de la catégorie par défaut. En revanche, seules les catégories créées par l'utilisateur peuvent être supprimées. Ceci a été mis en place pour que lorsqu'on supprime une catégorie liée à une ou plusieurs transactions, la catégorie soit remplacée par celle par défaut.

### 4.5 Transactions

Les transactions regroupent toutes les entrées et sorties d'argent.

L'utilisateur peut ajouter des revenus ou des dépenses. Chaque transaction est définie par un montant, une catégorie, un compte affecté (par défaut le compte principal sélectionné par l'utilisateur) et la devise utilisée (seuls les CHF sont pris en compte).

Les transactions peuvent être modifiées ou supprimées si besoin.

## 4.6 Dettes

L'utilisateur a la possibilité d'enregistrer ses dettes, qu'il en soit le débiteur ou le créancier. Il y a deux types de dettes, les dettes simples, qui ne sont qu'une information pour l'utilisateur, et les dettes synchronisées, qui lient deux utilisateurs de l'application.

Chaque dette possède un montant, une date limite et une description. Pour les dettes synchronisées, il est possible de spécifier un nom d'utilisateur.

Une fois une dette ou créance acquittée, son créateur peut la valider et la transaction qui en découle est automatiquement ajoutée. Dans le cas de dettes synchronisées, deux transactions sont créées, une pour chaque utilisateur.

## 4.7 Vue globale

L'utilisateur peut en tout temps suivre l'évolution de son compte principal (compte par défaut) grâce à une vue globale de toutes ses transactions pour le mois courant. Le graphique en courbe permet de comparer de manière graphique les différentes dépenses et rentrées d'argent alors que le diagramme circulaire affiche les dépenses par catégorie.

## 4.8 Budget

Un budget est forcément ponctuel (en fonction de deux dates données début/fin). Il permet de voir l'évolution des dépenses dans des catégories données. Les budgets sont représentés de manière graphique sur une ligne, avec la somme dépensée, le plafond et une ligne représentant le pourcentage atteint. L'utilisateur a également la possibilité de créer des budgets partagés, où les dépenses de chaque utilisateur sont communes et le budget évolue en fonction des participants (par exemple un budget de voyage ou de commissions pour la famille).

Un budget possède un nom, un montant et des catégories (aucune, une ou plusieurs). L'utilisateur peut modifier le montant, la durée et les catégories d'un budget.

Il est possible de supprimer n'importe quel budget, mais les conséquences sont différentes selon le cas. Le budget est supprimé définitivement si l'utilisateur en est le créateur, et que le budget est non partagé, ou partagé mais sans participants. Si le créateur d'un budget partagé avec des participants supprime le budget, un autre participant prend la place du créateur et ce dernier est retiré du budget. Si un participant du budget partagé supprime le budget, il est juste retiré des participants.

## 4.9 Budgets partagés

Pour les budgets partagés, l'utilisateur peut inviter d'autres utilisateurs (via leur nom d'utilisateur) à rejoindre le budget. Les personnes sont automatiquement ajoutées au budget sans confirmation.

N'importe quel utilisateur d'un budget partagé peut décider d'en sortir. Le créateur d'un budget partagé peut décider d'ajouter ou de supprimer des membres participants.

## 5 Conception et architecture du projet

### 5.1 Technologies utilisées

#### 5.1.1 Langage de programmation et librairie GUI

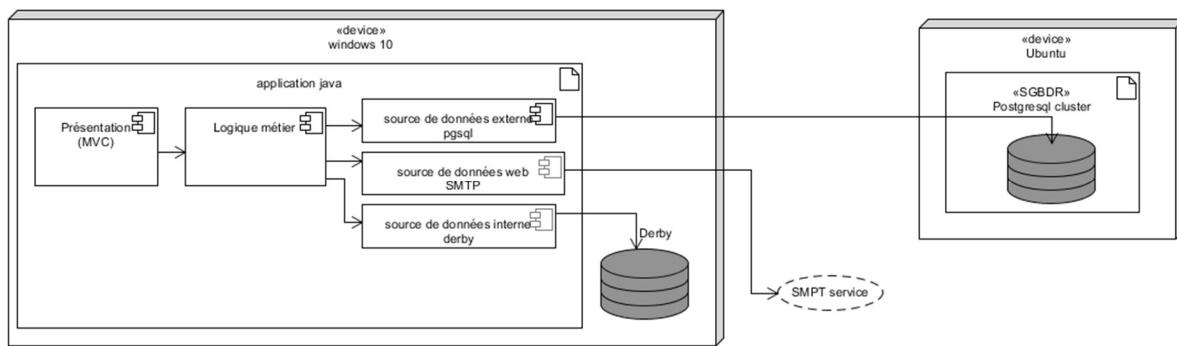
L'application a été implémentée en Java 9 avec l'environnement IntelliJ IDEA. Nous avons utilisé JavaFX pour son graphisme, afin de pouvoir séparer de manière claire la couche purement graphique de celle qui implémente les fonctionnalités graphiques, comme les boutons.

#### 5.1.2 Framework

Nous avons choisi d'utiliser Hibernate. Il s'agit d'un ORM (Object Relational Mapping) qui permet de développer des applications qui peuvent aisément gérer et accéder à des bases de données pour récupérer, modifier et supprimer des données. Hibernate a été utilisé pour réaliser la couche d'accès aux données internes (Derby) et la couche d'accès aux données externes (PostgreSQL). Ces couches d'accès aux données sont responsables de la persistance dans un système de gestion de base de données relationnel.

L'utilisation d'Hibernate nous a permis d'écrire du code plus facilement maintenable et compréhensible ainsi que de gérer les mises à jour et les changements sur plusieurs relations. Avec Hibernate, concevoir une couche capable de gérer la persistance des données est moins problématique et fastidieux que de le faire entièrement à la main.

### 5.2 Architecture de la solution



Architecture 1 - Schéma général de l'architecture de MoneyThoring

L'architecture est divisée en deux tiers :

1. Le serveur de base de données («VM» Ubuntu) avec PostgreSQL
2. Les clients (« device » Windows 10) sur lesquels l'application java est déployée

Le premier tiers est le serveur de base de données PostgreSQL qui fait office de base de données centralisée et s'utilise quand les utilisateurs travaillent directement en ligne. Ce serveur est une machine virtuelle Ubuntu. Le deuxième tiers est l'application cliente présente sur chacun des postes clients Windows 10.

L'application cliente est découpée en trois couches, chaque couche a une responsabilité précise :

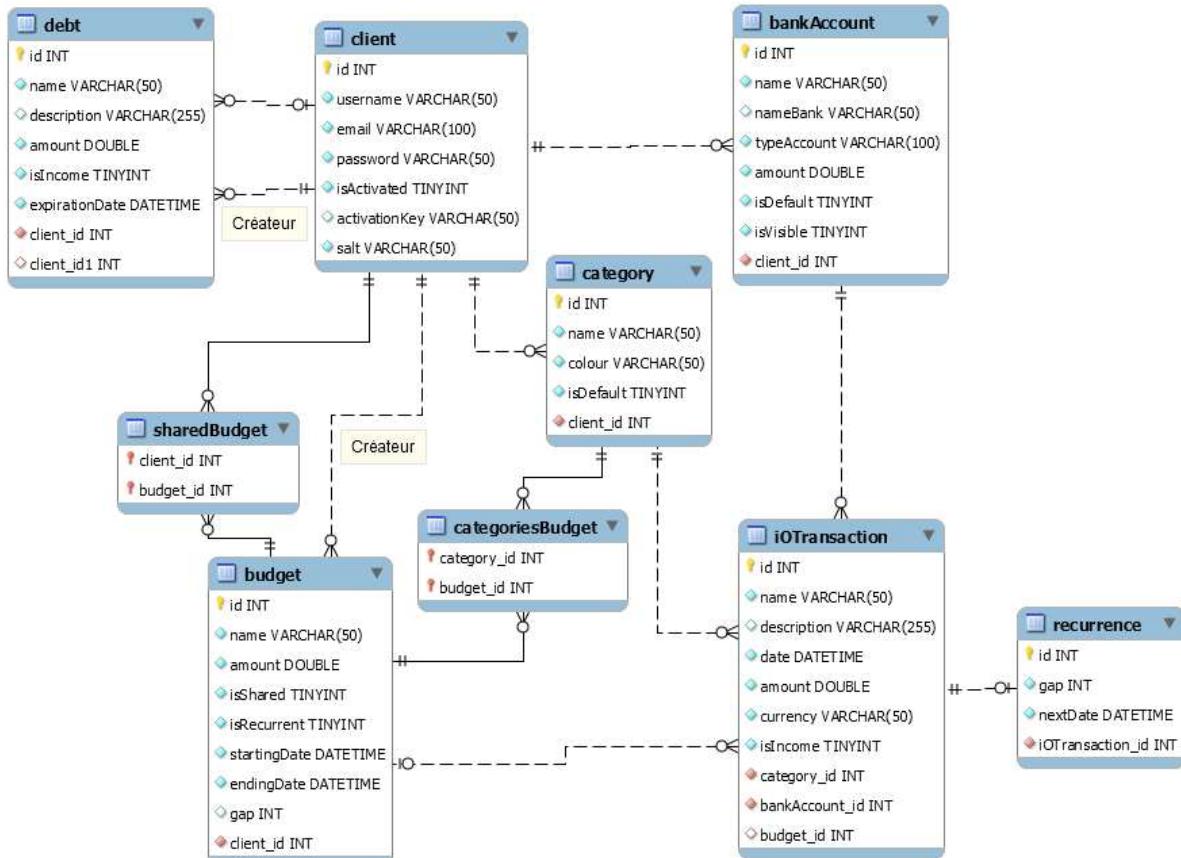
- La couche présentation
- La couche logique métier
- La couche d'accès aux données

Dans le projet, ces trois couches sont placées dans trois packages différents qui sont respectivement appelés :

1. GUI (Graphique User Interface) qui gère les interactions application-utilisateur et la présentation des données à l'aide d'interfaces graphiques.
2. BLL (Business Logic Layer) qui gère la logique applicative.
3. DAL (Data Access Layer) qui gère la persistance des données.

## 6 Bases de données (DB)

### 6.1 Schéma relationnel



DB 1 - Schéma relationnel de la base de données

La base de données de MoneyThoring, comme nous l'avons implémentée, possède deux acteurs majeurs : « client » et « budget ». A eux deux, ils représentent la majorité des interactions qu'il y a au sein de la base de données. Pour mieux comprendre notre projet, voici une description des diverses entités et interactions.

Un client possède un nom d'utilisateur (qui permettra aux autres utilisateurs de facilement le retrouver pour l'ajouter à un budget partagé), une adresse email, un mot de passe ainsi qu'un sel pour l'authentification, et une clé d'activation pour la création de son compte. Tant que le client n'a pas fait vérifier son adresse email lors de la création de son compte, il continue d'exister en base de données mais en tant qu'utilisateur « non activé ». Une fois la vérification faite, il peut sans autre se connecter et être reconnu en tant qu'utilisateur actif.

Un client possède un ou plusieurs comptes bancaires. Ceux-ci ne sont que des informations et ne possèdent pas de lien permettant l'identification d'un compte réel. Ils possèdent un nom, un nom de banque, un type et un montant. Le client peut choisir un compte en tant que compte par défaut. Celui-ci sera automatiquement sélectionné dans le cas de transactions. Un compte bancaire n'est pas supprimable. Du point de vue de l'utilisateur, il peut effectivement être supprimé, mais au niveau de la base de données il continuera d'exister en tant que compte invisible. Ce choix a été fait pour garder une trace des différentes transactions associées.

Un client peut créer deux types de dettes. La première étant une dette où c'est lui qui doit de l'argent à une entreprise ou une personne quelconque. Dans la deuxième, des personnes lui doivent de l'argent. Dans le cas où ces personnes seraient des utilisateurs de l'application, il peut insérer leur nom de compte et ainsi leur envoyer la dette créée.

Deux types de budgets peuvent être créés. Soit un utilisateur souhaite se faire un budget personnel, et dans ce cas il doit renseigner les catégories de transactions à prendre en compte dans le budget. Soit il fait partie ou crée un budget partagé et invite d'autres utilisateurs à le rejoindre.

Un budget est ponctuel, de ce fait il faut spécifier une date de début et une date de fin.

Les catégories possèdent un nom, une couleur et sont par défaut ou non. Une catégorie par défaut est une catégorie créée par le système et non par l'utilisateur et qui ne peut pas être supprimée (ex : non-catégorisé). Toutes les transactions concernées par la suppression d'une catégorie propre à l'utilisateur verront leur catégorie changer en « non-catégorisé ». Cette action est effectuée à l'aide d'un trigger sur la base de données.

Les transactions peuvent être soit des entrées, soit des sorties d'argent.

## 6.2 PostgreSQL

### 6.2.1 Hébergement de la base de données

Les recherches menées pour obtenir une instance de base de données accessible de façon centralisée ont pris beaucoup de temps. Nous avons fait des recherches sur des sites comme Ovh qui fournit un VPS (serveur virtuel privé) accessible depuis internet. Le problème est que les offres de ce site ne sont pas suffisamment détaillées et les caractéristiques techniques insuffisantes.

Nous avons pris une année de test chez Amazon pour un serveur cloud, afin d'héberger une base de données centralisée. Malheureusement le serveur se trouve en Amérique et de ce fait les temps de latence sont considérables.

Nous avons également ouvert un ticket chez Infomaniak.ch pour obtenir une machine sur le cloud. Ceux-ci nous ayant fait une offre gratuite uniquement pour un mois, nous avons dû décliner.

Comme nous ne développons pas de services pour le logiciel, il était impératif qu'un serveur dans l'intranet de l'école ou en Suisse soit accessible, car le trafic entre les clients et le serveur de base de données est conséquent et non optimisé via l'intermédiaire d'un service. Comme nous n'accédons pas aux données à travers un service web, mais directement avec un Object Relational Mapping intégré dans les clients, nous ne pouvons pas optimiser le trafic de données et limiter la quantité d'informations qui transite entre le serveur et le client.

### 6.2.2 Script de création

Le script PostgreSQL a été créé à partir du script SQL, généré depuis le schéma relationnel réalisé sur MySQL. Nous avons choisi de simplifier le script au maximum pour ne pas avoir des soucis d'adaptation entre les deux systèmes de base de données. Etant donné que notre schéma de base de données ne comporte pas de relations 1:1 à 1:1, 1:1 à 1:N et 1:N à 1:N, nous n'avons pas non plus besoin d'activer/désactiver des contraintes temporaires. Ceci explique le fait que nous ne renommions pas ces relations, puisque nous n'allons pas travailler dessus.

### 6.3 Derby

Puisque notre idée de base était d'utiliser SQLite comme base de données locale, nous avons installé un browser SQLite pour générer la base de données d'après le script SQL du schéma et la tester avec Hibernate et JPA.

Le browser SQLite que nous avons utilisé peut se télécharger à l'adresse suivante : <http://sqlitebrowser.org/>

À la suite de l'utilisation de SQLite avec Hibernate, nous avons rencontré plusieurs problèmes pendant la mise en place du projet. Ces derniers provenaient essentiellement des librairies qui ne supportaient pas la base de données. Après avoir consulté la page web <https://stackoverflow.com/questions/17587753/does-hibernate-fully-support-sqlite>, nous avons choisi de prendre Derby à la place de SQLite pour notre projet, celui-ci étant développé en Java.

### 6.4 Difficultés rencontrées

Globalement, les seules difficultés rencontrées au niveau des bases de données sont liées à la syntaxe des requêtes de PostgreSQL et Derby. Étant deux types de bases de données que nous ne connaissons pas, puisque nous avons toujours appris et utilisé des requêtes de type SQL, nous avons dû beaucoup nous renseigner pour traduire les scripts de création des bases ainsi que les différentes requêtes et autres triggers/fonctions en PostgreSQL et Derby.

## 7 Couche d'accès aux données (DAL)

### 7.1 Projet Maven

La création d'un projet avec Maven nous a pris du temps. Nous avons dû importer toutes les dépendances, afin qu'Hibernate puisse fonctionner correctement avec la version 9 de Java. Les premiers projets de tests n'étaient pas faits avec Maven, pour comprendre comment configurer les mappings et Hibernate. Comme nous avons deux sources de données différentes, PostgreSQL et Derby, nous avons passé du temps à comprendre comment les intégrer au projet.

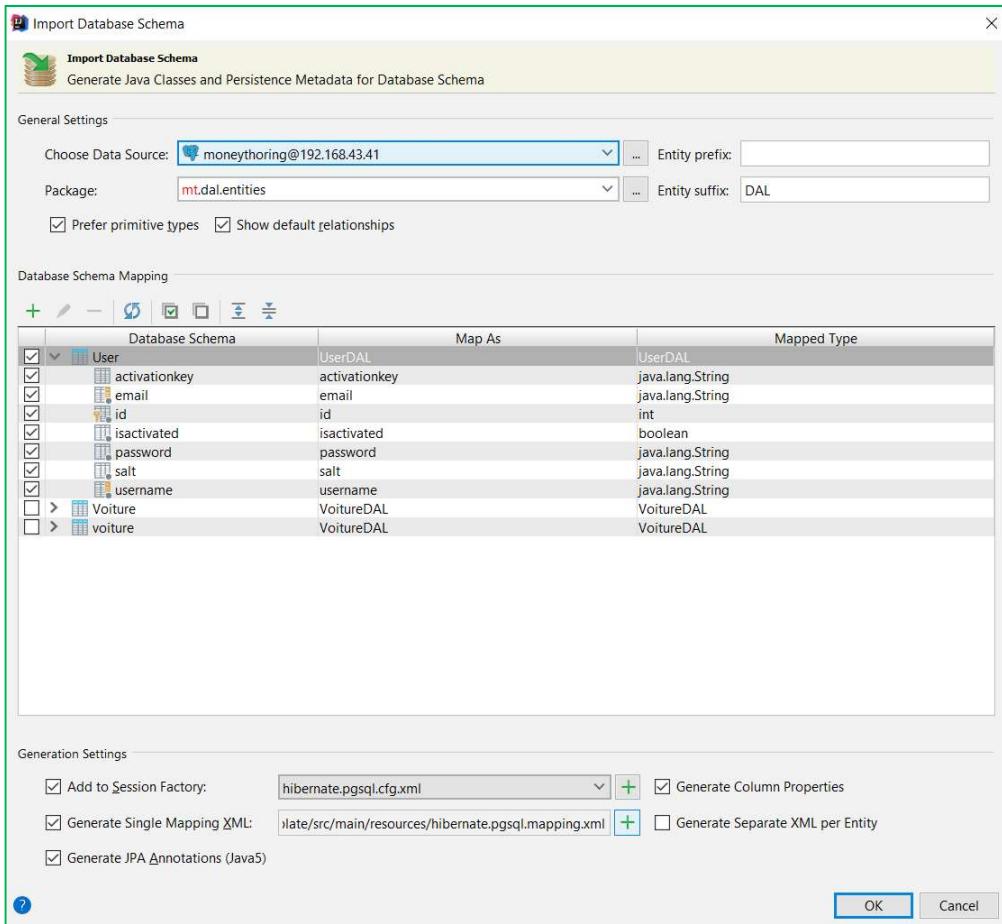
Avec NetBeans, l'intégration de la spécification JPA et Hibernate se fait beaucoup plus aisément que sur IntelliJ IDEA, notre environnement de développement. Ceci s'explique par le fait que NetBeans mette directement tous les fichiers de mapping et de configuration aux bons endroits avec toutes les dépendances nécessaires à la bonne utilisation d'Hibernate.

Nous avons donc créé ce projet pour Hibernate sur IntelliJ IDEA, afin d'avoir les bonnes dépendances et configurations. La première chose que nous avons essayé de réaliser sur ce projet est l'utilisation et la gestion d'un même schéma de base de données sur deux bases de données différentes. Ce qui signifie utiliser le mapping d'un même modèle d'entités sur deux systèmes de base de données différents. Pour cela nous avons installé ces deux systèmes, PostgreSQL et Derby. Puis nous avons exécuté le même script de création de tables.

Pour la création de ce projet nous avons fait un nouveau projet Maven appelé mvn\_project. Ensuite dans « project structure » nous avons ajouté un nouveau module nommé Hibernate, dont la librairie a été téléchargée depuis l'utilitaire d'ajout de module. Dans le but de configurer et générer les classes à partir de la base de données, nous avons utilisé l'outil de persistance fourni dans IntelliJ IDEA (attention : cet outil ne fait pas tout automatiquement). Avec l'outil de persistance nous avons généré le modèle depuis le schéma de bases de données.

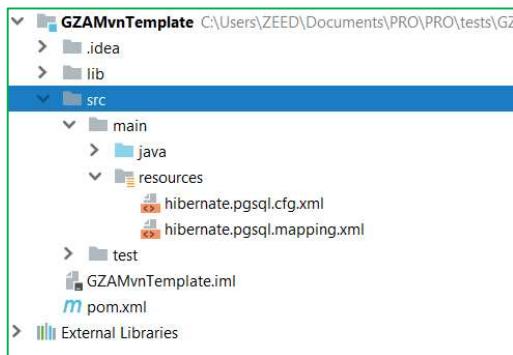
Dans le dossier de ressources nous avons placé un fichier de mapping et le fichier de configuration pour Hibernate :

- « hibernate.pgsqI.cfg.xml » pour le fichier de configuration d'Hibernate pour PostgreSQL
- « hibernate.pgsqI.mapping.xml » pour le fichier de mappage des entités d'Hibernate pour PostgreSQL



DAL 1 - Importation du schéma de base de données PostgreSQL

Une fois la configuration effectuée pour PostgreSQL, nous avons une arborescence de projet comme celle ci-dessous :



DAL 2 - Arborescence de projet après la configuration DB de PostgreSQL

Pour donner suite à cela, nous avons procédé à un premier test d'Hibernate, bien que nous n'eussions pas encore toutes les librairies nécessaires au bon fonctionnement du projet. Nous avons ajouté un fichier de mapping d'entité pour JPA pour tester le tout, mais avant nous avons juste testé avec Hibernate le code de la figure DAL 3, qui utilise une session.

```

public class Main {
    private static final SessionFactory ourSessionFactory;

    static {
        try {
            Configuration configuration = new Configuration();
            configuration.configure();

            ourSessionFactory = configuration.buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static Session getSession() throws HibernateException {
        return ourSessionFactory.openSession();
    }

    public static void main(final String[] args) throws Exception {
        final Session session = getSession();
        try {
            Query query = session.createQuery("from mt.dal.entities.ClientEntity");
            List listes = query.list();
            Iterator iterator = listes.iterator();
            while (iterator.hasNext()) {
                ClientEntity client = (ClientEntity) iterator.next();
                System.out.println(client.getEmail());
            }
        } finally {
            session.close();
        }
        System.out.println("End");
    }
}

```

DAL 3 - Code de test de l'intégration d'Hibernate

Une fois ce code exécuté, plusieurs librairies ont dû être importées avec Maven pour éviter des problèmes de compatibilité et pour ajouter des dépendances manquantes.

Voici ce que nous avons obtenu lors de la première exécution de ce code :

```

Run Main
" C:\Program Files\Java\jdk-9.0.4\bin\java" ...
java.lang.ExceptionInInitializerError
    at Main.<clinit> (Main.java:21)
    Caused by: java.lang.NoClassDefFoundError: javax/xml/bind/JAXBException <4 internal calls>
        at Main.<clinit> (Main.java:16)
        Caused by: java.lang.ClassNotFoundException: javax.xml.bind.JAXBException
            at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass (BuiltinClassLoader.java:582)
            at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass (ClassLoaders.java:185)
            at java.base/java.lang.ClassLoader.loadClass (ClassLoader.java:496)
            ... 5 more
Exception in thread "main"
Process finished with exit code 1

```

DAL 4 - Résultat de l'exécution du premier test d'Hibernate

Pour résoudre cette exception, nous avons importé les librairies « javax.xml.bind » et « com.sun.xml.bind », et ajouté le contenu de la figure DAL 5 au pom.xml.

```
<!-- https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api -->
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/com.sun.xml.bind/jaxb-core -->
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-core</artifactId>
    <version>2.3.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/com.sun.xml.bind/jaxb-impl -->
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>2.3.0</version>
</dependency>
```

DAL 5 - Dépendances Maven ajoutées au pom.xml

Nous avons dû spécifier le fichier xml de configuration dans le « session config », car nous utilisons deux fichiers de configuration différents, un pour PostgreSQL et l'autre pour Derby.

Les erreurs de la deuxième exécution du code étaient issues d'un manque de configurations dans le fichier de configuration Hibernate. Celles-ci étaient principalement les propriétés telles que le nom d'utilisateur et le mot de passe de la base de données.

```
<property name="connection.username">username</property>
<property name="connection.password">mdp</property>
```

DAL 6 - Configurations d'Hibernate manquantes

Au troisième lancement du projet, nous avons obtenu l'erreur suivante :

```
"C:\Program Files\Java\jdk-9.0.4\bin\java" ...
avr. 04, 2018 6:23:48 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core (5.2.16.Final)
avr. 04, 2018 6:23:48 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
java.lang.ExceptionInInitializerError
    at Main.<clinit>(Main.java:21)
Caused by: java.lang.NoClassDefFoundError: javax/activation/DataSource
    at com.sun.xml.bind.v2.model.impl.RuntimeBuiltinLeafInfoImpl.<clinit>(RuntimeBuiltinLeafInfoImpl.java:478)
    at com.sun.xml.bind.v2.model.impl.RuntimeTypeInfoSetImpl.<init>(RuntimeTypeInfoSetImpl.java:63)
    at com.sun.xml.bind.v2.model.impl.RuntimeModelBuilder.createTypeInfoSet(RuntimeModelBuilder.java:128)
    at com.sun.xml.bind.v2.model.impl.RuntimeModelBuilder.createTypeInfoSet(RuntimeModelBuilder.java:84)
    at com.sun.xml.bind.v2.model.impl.ModelBuilder.<init>(ModelBuilder.java:162)
    at com.sun.xml.bind.v2.model.impl.RuntimeModelBuilder.<init>(RuntimeModelBuilder.java:92)
    at com.sun.xml.bind.v2.runtime.JAXBContextImpl.getTypeInfoSet(JAXBContextImpl.java:455)
    at com.sun.xml.bind.v2.runtime.JAXBContextImpl.<init>(JAXBContextImpl.java:303)
    at com.sun.xml.bind.v2.runtime.JAXBContextImpl.<init>(JAXBContextImpl.java:139)
    at com.sun.xml.bind.v2.runtime.JAXBContextImpl$JAXBContextBuilder.build(JAXBContextImpl.java:1156)
    at com.sun.xml.bind.v2.ContextFactory.createContext(ContextFactory.java:165)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:564)
    at javax.xml.bind.ContextFinder.newInstance(ContextFinder.java:297)
    at javax.xml.bind.ContextFinder.newInstance(ContextFinder.java:296)
    at javax.xml.bind.ContextFinder.find(ContextFinder.java:409)
    at javax.xml.bind.JAXBContext.newInstance(JAXBContext.java:721)
    at javax.xml.bind.JAXBContext.newInstance(JAXBContext.java:662) <5 internal calls>
    at Main.<clinit>(Main.java:17)
Caused by: java.lang.ClassNotFoundException: javax.activation.DataSource
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:582)
    at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:185)
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:496)
    ... 26 more
Exception in thread "main"
Process finished with exit code 1
```

DAL 7 - Résultat de l'exécution du troisième test d'Hibernate

Pour remédier à cette erreur, nous avons importé la librairie suivante dans le pom.xml et ajouté le dernier driver en date pour PostgreSQL :

```
<!-- https://mvnrepository.com/artifact/javax.activation/activation -->
<dependency>
    <groupId>javax.activation</groupId>
    <artifactId>activation</artifactId>
    <version>1.1.1</version>
</dependency>
```

DAL 8 - Dépendance Maven ajoutée au pom.xml

```
<!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.2</version>
</dependency>
```

DAL 9 - Mise à jour du driver utilisé pour PostgreSQL

Une fois toutes ces modifications effectuées, le projet s'est enfin lancé. Voici donc le résultat de l'exécution du programme :

```
totot@hotmail.com
End
```

Par la suite nous avons réitéré ces opérations pour ajouter et faire fonctionner Hibernate avec Derby.

Voici les configurations finales des différents fichiers cités précédemment :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>MoneyThoring</groupId>
    <artifactId>MoneyThoring</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api -->
        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>2.3.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/com.sun.xml.bind/jaxb-core -->
        <dependency>
            <groupId>com.sun.xml.bind</groupId>
            <artifactId>jaxb-core</artifactId>
            <version>2.3.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/com.sun.xml.bind/jaxb-impl -->
        <dependency>
            <groupId>com.sun.xml.bind</groupId>
            <artifactId>jaxb-impl</artifactId>
            <version>2.3.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/javax.activation/activation -->
        <dependency>
            <groupId>javax.activation</groupId>
            <artifactId>activation</artifactId>
            <version>1.1.1</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>42.2.2</version>
        </dependency>
    </dependencies>
</project>
```

DAL 10 - pom.xml

```

<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property
      name="connection.url">jdbc:postgresql://192.168.43.41:5432/moneythoring</property>
    <property name="connection.driver_class">org.postgresql.Driver</property>
    <property name="connection.username">postgres</property>
    <property name="connection.password">Google2018$</property>
    <mapping class="mt.dal.entities.VoitureEntity"/>
    <mapping resource="hibernate.pgsqI.mapping.xml"/>

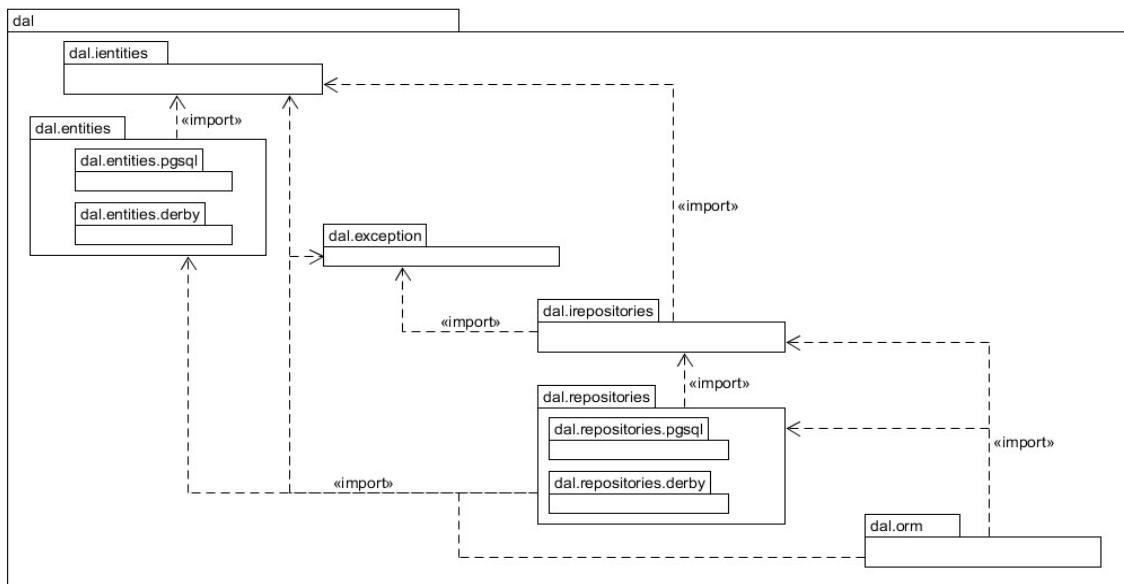
    <!-- DB schema will be updated if needed -->
    <!-- <property name="hbm2ddl.auto">update</property> -->
  </session-factory>
</hibernate-configuration>

```

DAL 11 - hibernate.pgsqI.cfg.xml

## 7.2 Mise en place de la DAL

La couche d'accès aux données encapsule les mécanismes propres à la persistance des données et permet de manipuler aisément les données stockées dans PostgreSQL et Derby. Voici la structure des packages de la couche d'accès aux données.



DAL 12 - Structure de la couche d'accès aux données

### 7.2.1 dal.dalexception

Le package « dalexception » contient les exceptions que la couche d'accès aux données renvoie aux couches supérieures. Ainsi nous pouvons encapsuler toutes les exceptions générées par Hibernate et les différentes parties du code de la couche d'accès aux données. Depuis ce package, nous pouvons également définir les traitements à appliquer en cas d'exception.

### 7.2.2 dal.entities

Ce package regroupe toutes les interfaces que les entités Derby et PostgreSQL implémentent. Ces interfaces sont utilisées par les couches supérieures pour manipuler les objets.

### 7.2.3 dal.entities

Ce package rassemble toutes les entités utilisées par Hibernate lors du mapping. Il contient deux sous-dossiers liés aux bases de données, l'un regroupe les entités pour le mapping de Derby et l'autre les entités pour le mapping de PostgreSQL. Chaque entité dans ces deux packages implémente toutes les interfaces qui se trouvent dans le package « entities ». Nous avons fait ce choix pour factoriser plus facilement le code et exposer directement des interfaces à mapper aux couches supérieures. Ainsi, les couches supérieures n'ont pas besoin de connaître l'implémentation des entités qu'elles manipulent et de savoir si ce sont bien des entités Derby ou PostgreSQL. Pour le mappage, les couches supérieures utilisent simplement un mapper, qui convertit un objet d'un modèle d'entité provenant de la couche supérieure en le mappant soit à un objet du modèle d'entité Derby ou du modèle d'entité PostgreSQL.

#### 7.2.3.1 dal.entities.derby

Ce package contient l'implémentation des interfaces dal.entities pour le mapping objet Derby. Le modèle objet Derby est tagué avec les annotations d'Hibernate, afin de mapper cet objet au modèle de base de données Derby. Donc à chaque implémentation correspond une équivalence sous forme de table en base de données selon la structure, la configuration du fichier de mapping Hibernate et des annotations.

#### 7.2.3.2 dal.entitiespgsql

Idem que pour Derby, ce package contient l'implémentations des interfaces dal.entities pour le mapping objet PostgreSQL. Le modèle objet PostgreSQL est tagué avec les annotations d'Hibernate, afin de mapper cet objet au modèle de base de données PostgreSQL. Donc à chaque implémentation correspond une équivalence sous forme de table en base de données selon la structure, la configuration du fichier de mapping Hibernate et des annotations.

### 7.2.4 dal.repositories

Ce package contient les interfaces qui permettent d'effectuer des opérations en base de données tel que : création, modification, suppression ainsi que d'autres traitements comme la récupération de toutes les entités d'une association. Les opérations publiques dans les interfaces de ce package s'appliquent aussi bien pour Derby que pour PostgreSQL. C'est pour ces raisons que ces interfaces n'utilisent aucune implémentation, mais uniquement les interfaces contenues dans dal.entities.

#### 7.2.4.1 dal.repository.derby

Ce package regroupe les classes qui implémentent les interfaces du package dal.repository. Dans ce package, l'implémentation est uniquement réservée à Derby.

#### 7.2.4.2 dal.repositorypgsql

Comme pour les repository derby, ce package regroupe les classes qui implémentent les interfaces du package dal.repository. Il est uniquement question de l'implémentation propre à PostgreSQL.

### 7.2.5 dal.orm

Ce package contient en quelque sorte une unité de travail, chargée d'instancier une seul fois chaque répertoire, de les stocker et de les retourner pour pouvoir utiliser les méthodes d'accès aux données. L'objet qui regroupe tous ces répertoires est chargé de les instancier avec la

---

même session et de les faire travailler sur une même connexion. Cela permet de faire travailler tous les répertoires en harmonie sur une session partagée entre tous. Les logiques de commit, rollback et transaction sont à implémenter directement dans cet objet. Nous sommes forcés de le faire car Derby ne supporte pas plusieurs connexions simultanées.

### 7.2.6 Package de test

Nous avons aussi créé un package de test, afin de pouvoir s'assurer du bon fonctionnement et du bon comportement de la couche d'accès aux données lorsque nous appelons des méthodes. Ceci prend du temps mais semble indispensable, car nous mappons à chaque fois les objets aux couches plus hautes. Ce qui fait que la référence des objets qui ont été recherchés est à chaque fois perdue. Donc lors d'une modification d'un objet venant d'une couche plus haute, nous devons le convertir systématiquement en objet du modèle de la couche d'accès aux données. Les tests prennent également du temps, car nous souhaitons voir ce qu'il se passe avec le chargement paresseux et les propriétés de navigation au moment du mappage des objets entre la couche d'accès aux données et les couches plus hautes.

## 7.3 Hibernate

Avec Hibernate nous avons perdu du temps pour bien comprendre comment se configurait le mapping xml de chaque entité. Certains choix aussi ont dû se faire concernant les types primitifs ou des types plus évolués (leur équivalent en objet). Nous avons donc choisi de travailler principalement avec des types primitifs. Les autres choix concernent les types des propriétés de navigation, car ces propriétés doivent retourner les interfaces qu'implémente chaque classe, afin de pouvoir garder des méthodes communes aux modèles PostgreSQL et Derby.

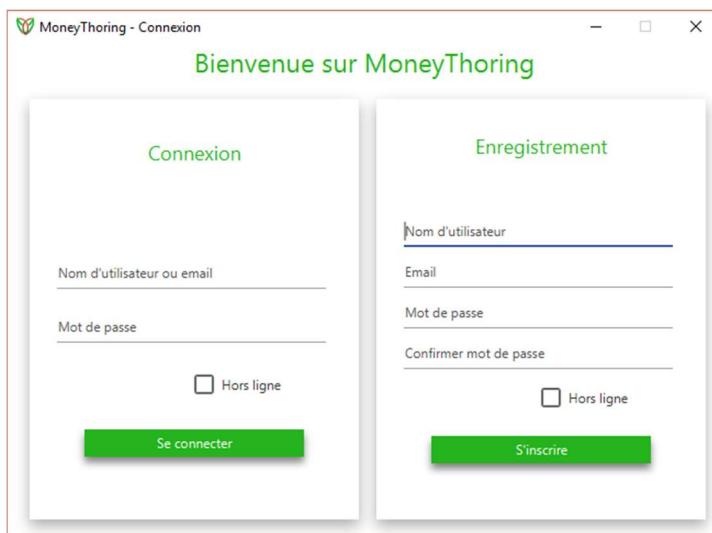
## 7.4 Difficultés rencontrées

Les difficultés rencontrées ont déjà été mentionnées précédemment.

## 8 Interface graphique (GUI)

Pour définir nos interfaces graphiques nous utilisons des fichiers au format FXML. C'est le format utilisé par JavaFX pour définir un squelette de fenêtre graphique. Ces fichiers utilisent la même syntaxe que les fichiers XML. JavaFX permet de dédier le contrôle de ces squelettes à des fichiers de type contrôleur. Dans notre implémentation, tous les contrôleurs implémentent l'interface « `Initializable` », qui permet de clairement séparer le FXML du contrôleur (aucune référence vers le contrôleur depuis le FXML).

### 8.1 Fenêtre de connexion/enregistrement d'un compte utilisateur



GUI 1 - Fenêtre de connexion à MoneyThoring

La fenêtre de connexion est la première fenêtre à laquelle seront confrontés tous les utilisateurs. Elle permet de créer un compte et de se connecter.

#### 8.1.1 Composition

Le conteneur principal de la fenêtre de connexion est un `Pane`, qui est la classe de base pour les panneaux de mise en page et permet simplement de positionner librement les éléments enfants. Nous en avons quatre : un label qui dispose le titre de notre vue, et trois `GridPane` pour l'enregistrement, la connexion et la confirmation du code de l'inscription (invisible dans la figure GUI 1).

On utilise des `GridPane` pour les espaces de connexion et d'enregistrement, car ils permettent d'ajouter des enfants dans une grille flexible. Un enfant peut être placé n'importe où dans la grille et peut s'étendre sur plusieurs lignes/colonnes. Cela nous facilite l'alignement des composants. Dans ces grilles, nous utilisons des `JFXTextField` ainsi que des `JFXPasswordFields` pour récupérer la saisie de l'utilisateur, et des `JFXButtons` pour la confirmation des saisies.

#### 8.1.2 Contrôleur

Nous avons un événement « `clickRegisterButton` », appelant la méthode « `register` » de la BLL, qui permet d'enregistrer un utilisateur dans notre base de données et d'envoyer un email contenant un code de confirmation à l'adresse fournie.

Un autre événement « `clickLoginButton` », appelant cette fois la méthode « `login` ». Cette dernière retourne une erreur dans le cas où aucune correspondance n'est trouvée entre les

données de la base de données et les informations saisies par l'utilisateur. Cela nous permet de relayer l'information à l'utilisateur grâce à un message.

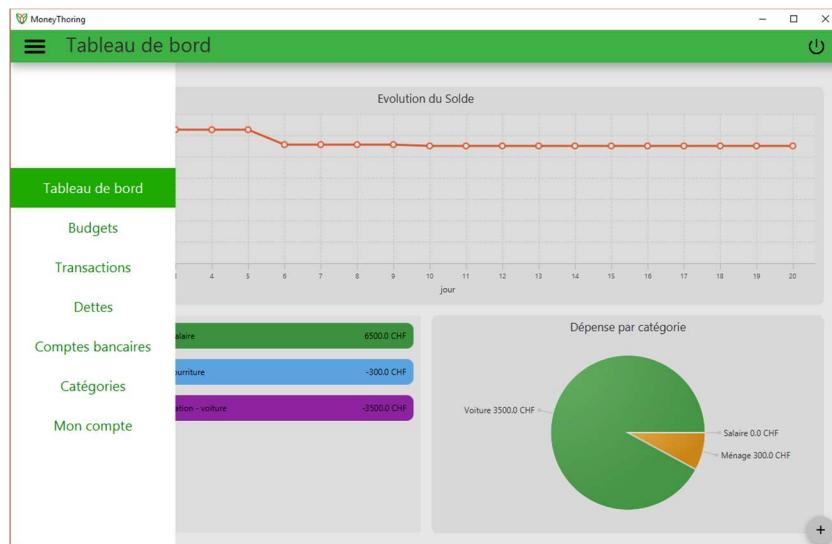
Enfin, nous avons un événement « confirmButton » qui va vérifier que le code saisi correspond au code généré. Cette opération n'est requise que lors de la première connexion de l'utilisateur.

Une fois connecté, nous faisons disparaître la fenêtre de connexion et créons la fenêtre principale.

## 8.2 Menu latéral de l'application

Le menu latéral est simplement l'outil qui permet de naviguer dans MoneyThoring. Il est composé d'une VBox, contenant les différents boutons permettant de choisir telle ou telle vue. Le contrôleur du menu latéral met simplement à jour les classes CSS des boutons pour indiquer lequel est sélectionné.

## 8.3 Fenêtre principale



GUI 2 - Fenêtre principale, menu ouvert

La fenêtre principale est la fenêtre que l'utilisateur final utilisera, afin de naviguer dans les différentes vues/fonctionnalités. Pour cette raison, elle demande un certain investissement pour la perfectionner.

### 8.3.1 Composition

Voici comment le fichier FXML de cette fenêtre est découpé :

A la racine, nous trouvons un *BorderPane*, afin de pouvoir séparer convenablement l'entête (au TOP NORD) du contenu de la fenêtre (CENTRE). Ces deux parties de notre *BorderPane* contiennent un *AnchorPane* pouvant fixer leur contenu :

- TOP : le top est composé d'un *JFXHamburger*, qui permettra d'ouvrir le menu latéral, et d'un label qui indique simplement "où" nous sommes dans le programme (p.ex. : "Tableau de bord", "Budget – Nourriture", etc.)
- CENTRE : la partie centrale du *BorderPane* contient un second *AnchorPane* qui permet de charger le contenu voulu (par exemple : la vue du Tableau de bord ou la liste des

budgets). Le centre contient également un *JFXDrawer*, qui est le conteneur de notre menu latéral.

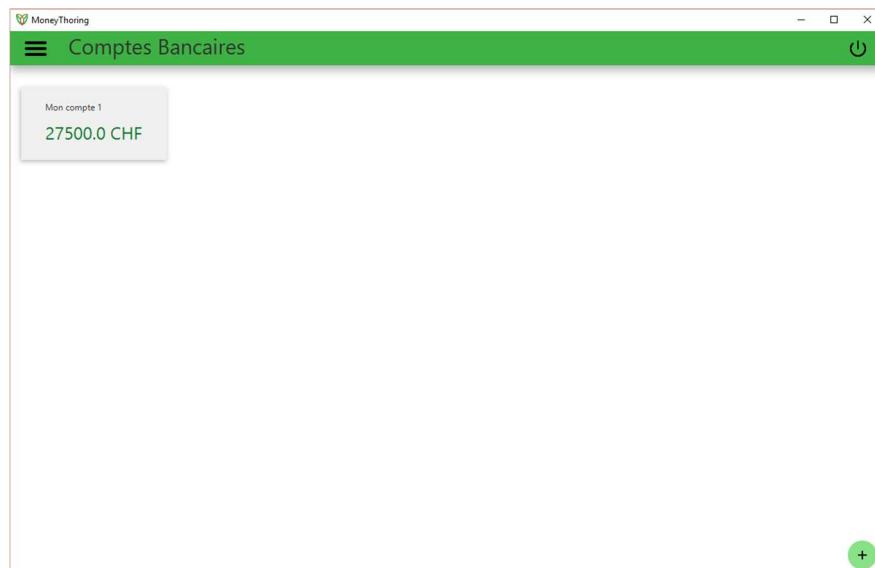
### 8.3.2 Contrôleur

Le contrôleur de la fenêtre principale permet d'ouvrir et fermer le menu latéral, mais surtout de charger les différentes vues (tableau de bord, comptes bancaires, ...). Afin de pouvoir gérer au mieux les transitions entre les vues, nous avons défini deux tableaux de String, un contenant le titre des vues, et l'autre gardant les chemins vers les fichiers FXML utilisés. Cependant, lors du chargement, nous ne pouvons pas nous passer de l'utilisation d'un switch-case, car nous devons définir le contrôleur des vues au moment du chargement.

## 8.4 Comptes bancaires

Cette fenêtre permet d'avoir une vue d'ensemble sur l'état des comptes bancaires enregistrés par le client.

### 8.4.1 Vue principale (liste)



GUI 3 - Fenêtre des comptes bancaires

#### 8.4.1.1 Composition

Cette vue possède comme conteneur principal un *AnchorPane*. Ce conteneur nous permet de fixer le bord de ses enfants à une certaine distance de son propre bord. Cela est pratique pour le redimensionnement de notre fenêtre et de ses éléments.

Dans ce conteneur, nous avons trois enfants :

- Un *FlowPane* pour lister les comptes bancaires
- Un bouton pour la création d'un compte bancaire
- Et un second *AnchorPane* pour accueillir soit le détail d'un compte, soit le formulaire de création d'un compte.

Nous utilisons un *FlowPane* pour lister les comptes bancaires, car cet élément gère le positionnement de ses enfants de façon autonome (les uns à la suite des autres, sans affecter leur dimensions). Pour ce qui est du contrôle pour l'ajout d'un compte bancaire, nous utilisons simplement un *JFXButton*.

#### 8.4.1.2 Contrôleur

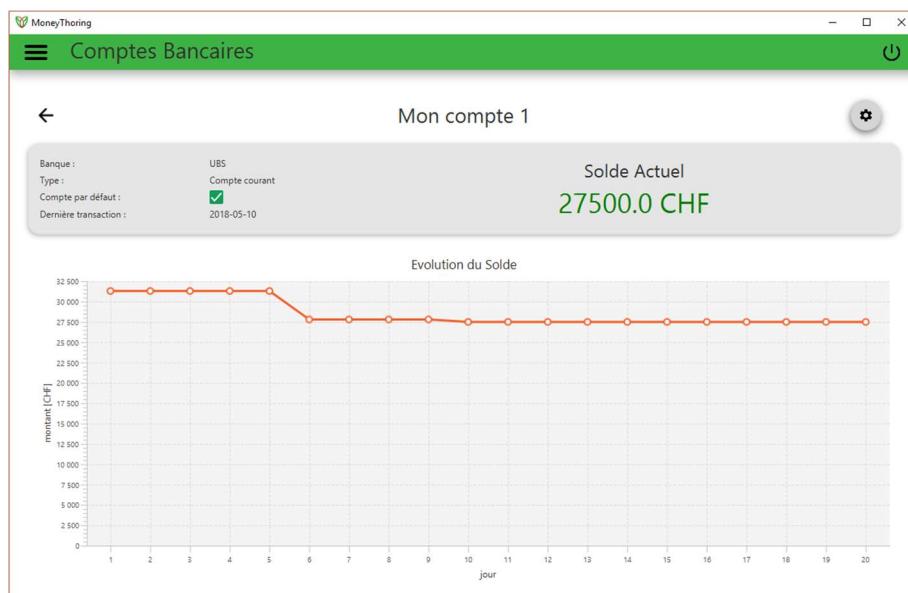
À l'initialisation de la fenêtre, nous récupérons la liste des comptes bancaires de l'utilisateur connecté afin de créer un « AccountDisplayer », pour chacun d'entre eux, que nous ajoutons à notre FlowPane.

Un clic sur notre bouton charge la vue de création d'un compte bancaire (le formulaire). Une fois le formulaire rempli, nous récupérons l'objet créé et l'affichons à l'aide d'un « AccountDisplayer ». Ce dernier est un objet qui permet de représenter graphiquement un compte bancaire, il est composé d'un GridPane, qui contient le nom attribué à notre compte bancaire et le montant de notre compte bancaire. Au clic d'un « AccountDisplayer », nous chargeons la vue du détail de notre compte bancaire.

#### 8.4.2 Vue détaillée

Cette vue permet de voir plus en détails les informations d'un compte bancaire. Nous pouvons également voir l'évolution de l'état du compte au fil de temps.

##### 8.4.2.1 Composition



GUI 4 - Vue détaillée d'un compte bancaire

Dans cette fenêtre, nous trouvons un LineChart qui nous permet de visualiser l'évolution du solde du compte bancaire, ainsi qu'un AnchorPane qui contient un résumé des propriétés du compte. Nous avons également un AnchorPane au premier plan qui sert à accueillir le formulaire des comptes bancaire (qui est, cette fois, utilisé pour modifier le compte en question).

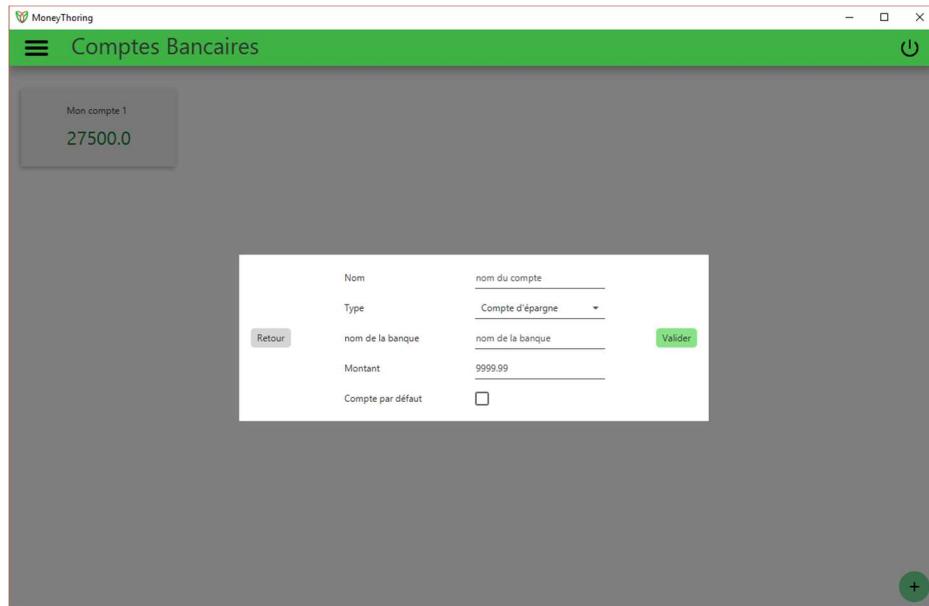
Pour la partie contrôle de cette vue, nous avons une JFXNodeList, qui nous permet de regrouper les boutons de modification et de suppression, et un bouton qui nous permet de retourner à la liste des comptes.

##### 8.4.2.2 Contrôleur

A l'initialisation du contrôleur nous récupérons les informations du compte sélectionné afin de les afficher et nous fixons les événements relatifs aux différents boutons (que nous créons et ajoutons à la JFXNodeList sur le moment).

La suppression du compte à proprement parler est gérée dans la liste des comptes. La modification, elle, appelle le formulaire et attend le retour de cette dernière pour mettre à jour les informations modifiées.

### 8.4.3 Formulaire des comptes bancaires



GUI 5 - Formulaire des comptes bancaires

Le formulaire des comptes bancaire permet de créer et modifier un compte bancaire.

#### 8.4.3.1 Composition

Toutes les vues de formulaires sont construites de la même façon :

L'élément racine est un `AnchorPane` qui possède un effet de transparence afin de pouvoir gérer les redimensionnements de la fenêtre. Puis nous trouvons une `HBox` contenant une `VBox`. L'utilisation de ces deux éléments sert à garder le contenu de la vue centrée en toute circonstance.

Puis nous avons un `AnchorPane` qui représente la zone du formulaire. Dans ce dernier nous avons placé les boutons d'annulation et de confirmation, ainsi que les champs du formulaire que l'utilisateur peut remplir.

#### 8.4.3.2 Contrôleur

Encore une fois, tous les contrôleurs de formulaires ont les mêmes fonctionnalités. Si le formulaire est appelé dans le but de créer un objet alors nous laissons les champs vides, sinon nous les remplissons avec les données de l'objet à modifier (ici un « `BankAccountLogic` »).

Lors de la confirmation, nous appelons une méthode du contrôleur qui pourra gérer la création/modification des objets.

## 8.5 Catégories

Les catégories sont une manière simple et intuitive de classer ses dépenses. Par défaut l'utilisateur en possède une, et peut en ajouter autant qu'il le veut.

### 8.5.1 Vue globale (liste)



GUI 6 - Liste des catégories

La vue de la liste des catégories permet à l'utilisateur de consulter ses catégories, d'en créer ou de les modifier si nécessaire.

#### 8.5.1.1 Composition

La liste des catégories est assez triviale. La racine est un *AnchorPane*, afin de pouvoir fixer la position du bouton d'ajout. Nous avons donc un bouton et un *FlowPane* qui nous permet d'afficher nos catégories de façon harmonieuse et sans effort. Nous avons également ajouté un *AnchorPane* pour accueillir le contenu du formulaire.

#### 8.5.1.2 Contrôleur

Lors de l'initialisation, nous récupérons la liste des catégories définies par l'utilisateur afin de pouvoir les afficher à l'aide d'un « displayeur ». Nous définissons également les événements du clic du bouton de création d'une catégorie et au clic du formulaire (modification d'une catégorie).

### 8.5.2 Formulaire de création d'une catégorie



GUI 7 - Formulaire des catégories (modification)

#### 8.5.2.1 Composition

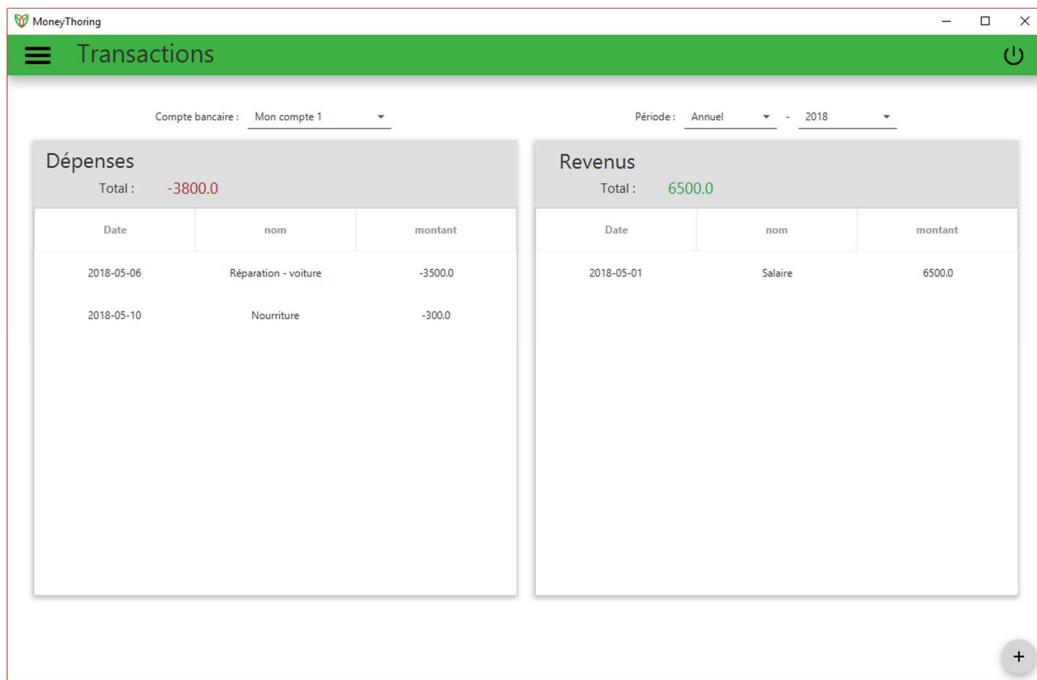
Une différence notable par rapport au formulaire précédent est la présence du bouton de suppression. En ce qui concerne le formulaire en lui-même, nous demandons un nom et une couleur pour une catégorie, nous utilisons donc les champs `JFXTextField` et `JFXColorPicker`.

#### 8.5.2.2 Contrôleur

Le bouton de suppression informe simplement la fenêtre appelante que l'utilisateur veut supprimer un élément. À part ce détail, le contrôleur de ce formulaire respecte l'implémentation du formulaire précédent.

## 8.6 Transactions

### 8.6.1 Vue globale



GUI 8 - Aperçu de la liste des transactions

La liste des transactions est une fenêtre plutôt importante, étant donné qu'elle permet d'avoir une vue d'ensemble sur l'activité d'un compte bancaire en fonction d'une période choisie.

### 8.6.1.1 Composition

La racine de la fenêtre est un *AnchorPane*, afin de pouvoir gérer le redimensionnement et la position de ses composants. Au sommet, nous avons des *ComboBox* pour choisir le compte bancaire et la période à prendre en compte. Au centre nous avons deux *BorderPane* (eux-mêmes contenu dans un *GridPane*), tous deux contiennent au centre une *JFXTreeView* qui permet de lister nos transactions.

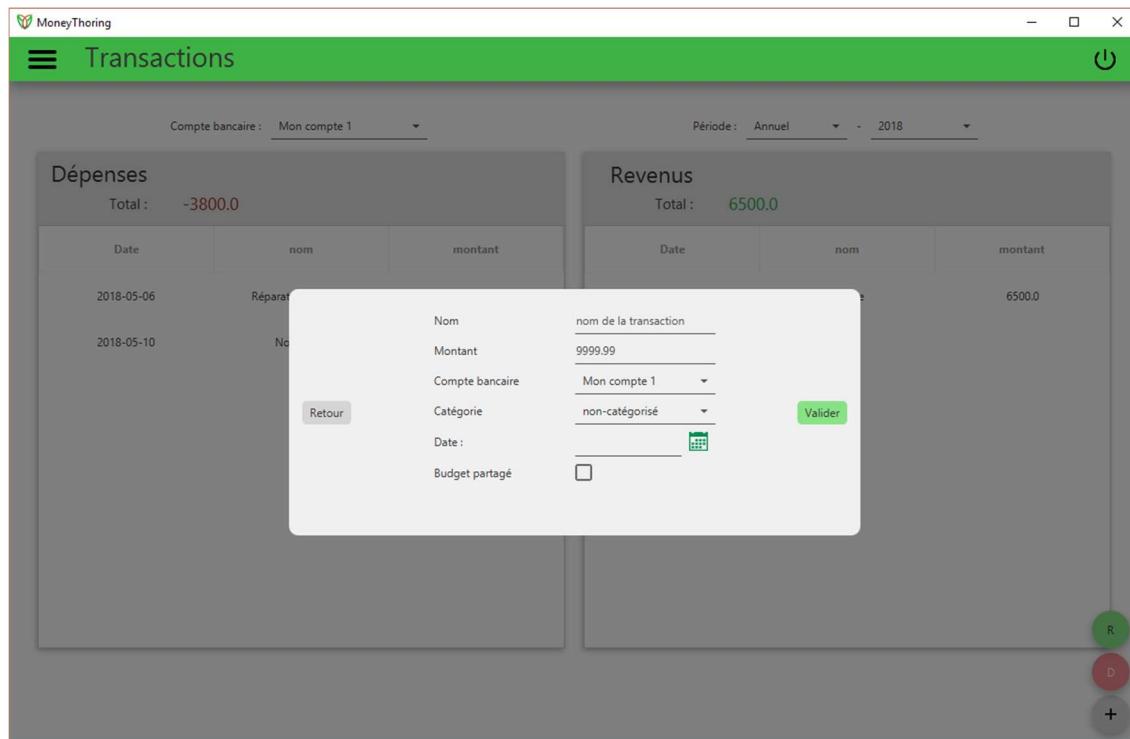
Pour l'ajout d'une dépense ou d'un revenu, nous utilisons une *JFXNodeList* qui permet d'avoir une liste de boutons (une pour les revenus et une pour les dépenses).

### 8.6.1.2 Contrôleur

Dans ce contrôleur, nous gérons tous les événements liés aux *ComboBox* (choix d'un compte bancaire et périodes). Lorsqu'un élément est sélectionné dans chacune des *ComboBox*, nous appelons la méthode « *setData* », qui elle va afficher les transactions liées au compte bancaire pour la période sélectionnée dans un de nos deux tableaux. Nous avons aussi deux événements sur les boutons de la *NodeList* pour appeler le formulaire de création d'une transaction.

Pour la modification/suppression d'une transaction, nous utilisons un événement au double clic sur la transaction, afin que le formulaire des transactions se charge.

## 8.6.2 Formulaire d'ajout/modification/suppression des transactions



GUI 9 - Formulaire des transactions (création)

### 8.6.2.1 Composition

La structure de cette fenêtre (sauf champs du formulaire) est la même que celle des catégories.

Le formulaire ici est contenu dans un *GridPane* dans lequel nous trouvons des *JFXTextfield* pour le nom et le montant de la transaction, des *JFXComboBox* pour le choix de la catégorie et le

choix du budget partagé, et un *JFXDatePicker* pour le choix de la date d'exécution de la transaction (nous vérifions que la date ne soit pas postérieure à la date actuelle).

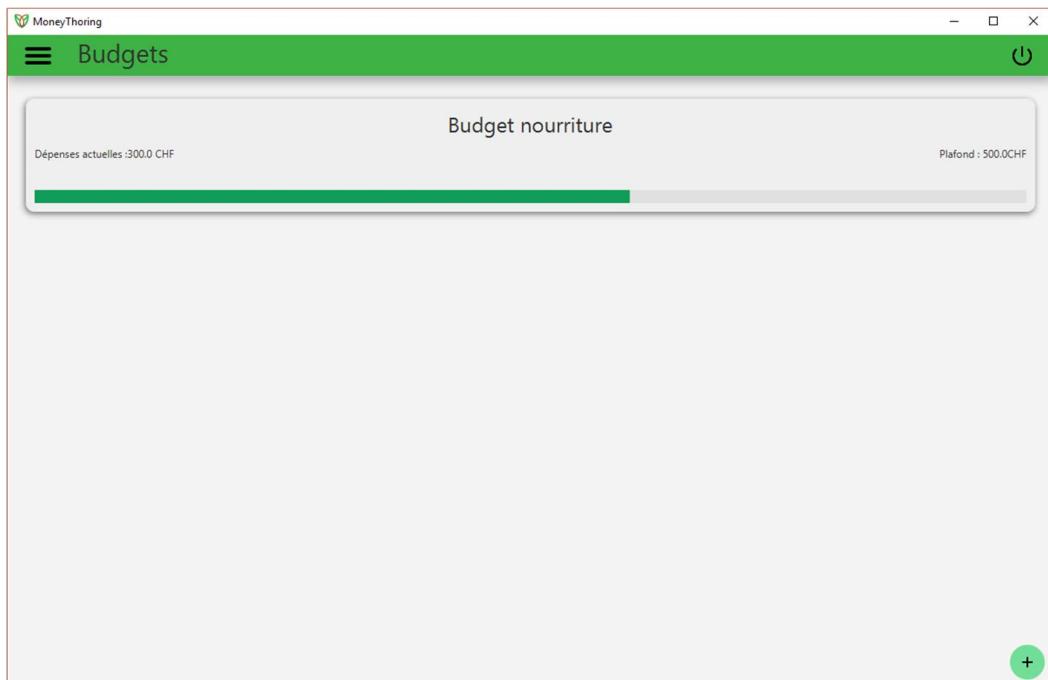
#### 8.6.2.2 Contrôleur

Ce contrôleur fonctionne de la même manière que le formulaire des catégories (annulation de la saisie, validation de la création/modification, suppression d'un objet).

## 8.7 Budgets

Les budgets sont une des principales fonctionnalités de notre projet. Son développement a pris un certain temps.

### 8.7.1 Vue globale



GUI 10 - Aperçu de la liste des budgets

La liste des budgets permet d'avoir une vision d'ensemble sur les budgets qui concernent l'utilisateur.

#### 8.7.1.1 Composition

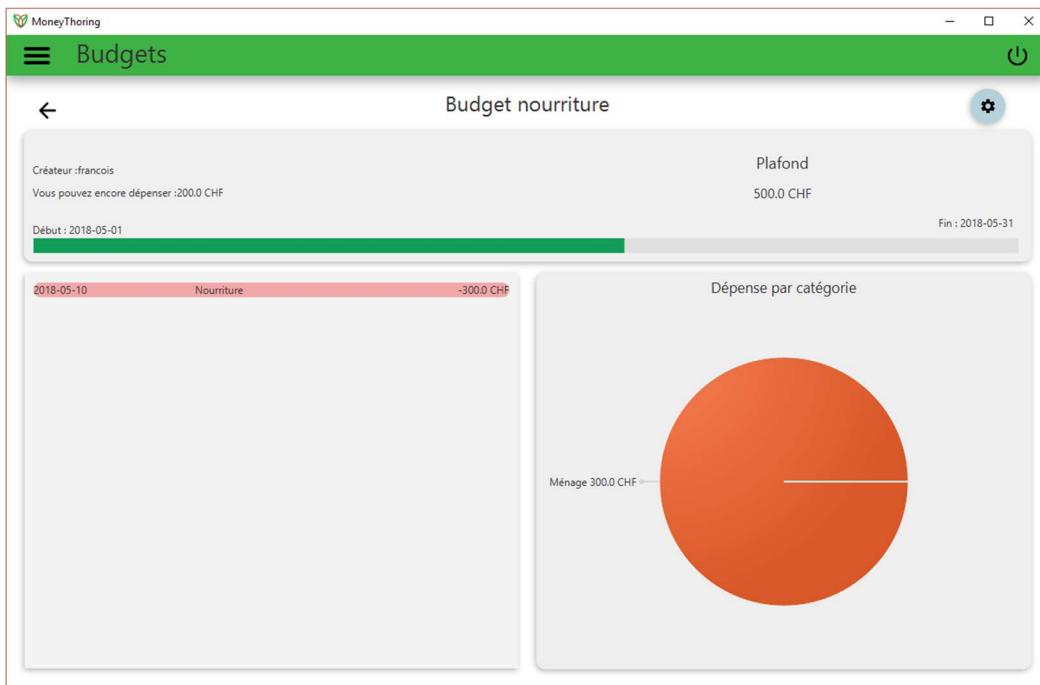
L'élément racine est un *AnchorPane*, afin de pouvoir redimensionner le contenu convenablement. A l'intérieur nous trouvons un *ScrollPane* qui possède une *VBox* pour faire en sorte que les budgets s'ajoutent verticalement (comme une liste).

Une différence notable par rapport aux autres "listes" est que le « Displayeur » utilisé n'est pas créé intégralement dans le code. Afin de pouvoir mieux gérer le positionnement des différents champs, nous avons créé un fichier « *budgetDisplay.fxml* ». L'élément racine est un *AnchorPane*, qui lui possède des labels pour le nom du budget ainsi que le plafond et la dépense actuelle et une *JFXProgressBar* pour l'évolution des dépenses du Budget.

### 8.7.1.2 Contrôleur

Le contrôleur de cette vue s'apparente un peu à celui des comptes bancaires, dans le sens où nous disposons d'une liste d'objets sur lesquels nous pouvons cliquer pour ouvrir un détail et d'un bouton pour en ajouter un.

### 8.7.2 Fenêtre du détail d'un budget



GUI 11 - Détail d'un budget

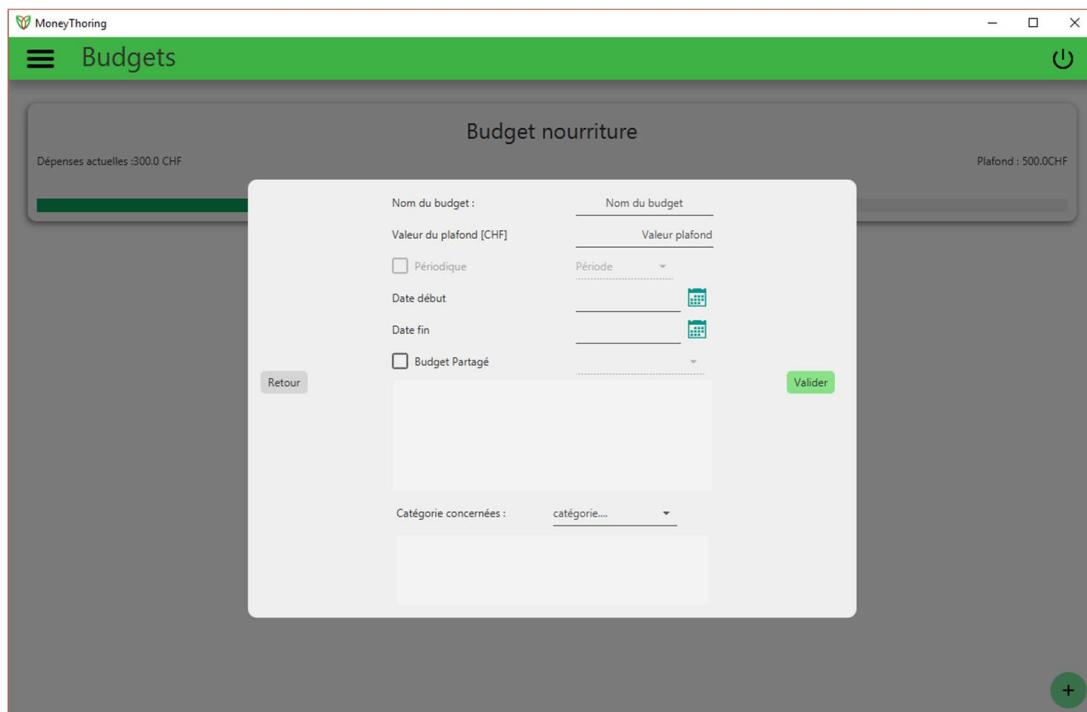
#### 8.7.2.1 Composition

A l'intérieur de l'élément racine (*AnchorPane*) nous trouvons un autre *AnchorPane* pour accueillir les informations générales du budget, comme le titre du budget, le créateur, ou encore la somme qu'il peut encore dépenser, le plafond, la date de début et de fin. Nous avons aussi un *GridPane* qui contient d'un côté un *ScrollPane* pour la liste des transactions et de l'autre un *AnchorPane* contenant un graphique en camembert (*PieChart*). Enfin nous avons, comme dans le détail des comptes bancaires, une *JFXNodeListe* pour afficher les boutons de modification et de suppression.

#### 8.7.2.2 Contrôleur

A l'initialisation de cette vue, nous récupérons les informations du budget dont on veut le détail pour les afficher dans les champs correspondants. Nous en profitons également pour récupérer et afficher toutes les dépenses liées aux catégories surveillées par le budget dans l'intervalle des dates prises en compte par le budget. Dans le cas où le budget ne surveille aucune catégorie, nous prendrons en compte les dépense de tous les comptes bancaires de l'utilisateur.

### 8.7.3 Formulaire d'un budget



GUI 12 - Formulaire de création d'un budget

#### 8.7.3.1 Composition

Le formulaire (figure GUI 12) consiste en des *Textfield* pour le nom et le montant du budget, deux *JFXDatePicker* pour la saisie de la date de début et de fin du budget. Nous avons aussi une *JFXCheckBox* qui nous permet de dire si le budget est partagé, si tel est le cas, nous pouvons lier des utilisateurs à ce budget via une *ComboBox*. Les utilisateurs s'affichent dans l'*AnchorPane* juste en dessous. Un mécanisme similaire est disponible pour les catégories : nous sélectionnons les catégories via une *ComboBox*, et elles s'affichent juste en dessous dans un *AnchorPane*.

#### 8.7.3.2 Contrôleur

Pour cette vue, outre les mécanismes de création/modification/suppression de budget, il nous a fallu également gérer la liste des utilisateurs et des catégories liées à ce dernier. Nous utilisons une *ArrayList* de « *UserModel* » respectivement « *CategoryLogic* » pour tenir à jour la liste des éléments sélectionnés. À la sélection d'un élément des *ComboBox*, nous ajoutons l'élément à la liste correspondante et créons également un "mini" « *display* » pour rendre l'information visuelle. On peut retirer un élément de ces listes simplement en cliquant sur la croix des « *displayers* ».

### 8.7.4 Tableau de bord

Le tableau de bord est la "page d'accueil" de MoneyThoring. C'est la vue par défaut présentée à la connexion de l'utilisateur.



GUI 13 - Tableau de bord

#### 8.7.4.1 Composition

L'élément racine (toujours un *AnchorPane*) possède un *GridPane*. Ce dernier contient trois *AnchorPane*. Deux d'entre eux affichent un graphique (un à courbe et l'autre en camembert). Le dernier possède un *ScrollPane* contenant une *VBox* pour lister les transactions du compte par défaut de l'utilisateur

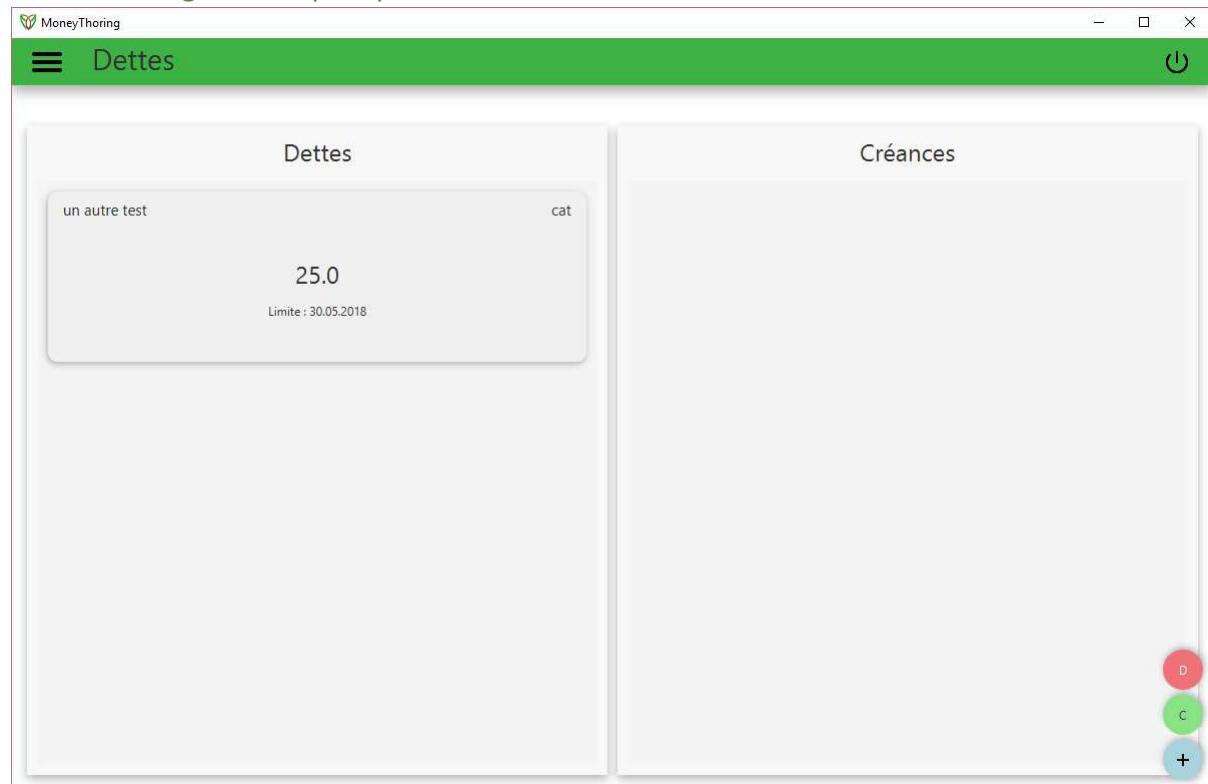
#### 8.7.4.2 Contrôleur

Lors de l'initialisation du tableau de bord, nous récupérons toutes les transactions du mois courant liées au compte par défaut. Le graphique en courbe permet de voir l'évolution dudit compte, alors que le diagramme en camembert représente la distribution des dépenses par catégories.

## 8.8 Dettes

La gestion des dettes et des créances est une fonctionnalité que nous trouvions intéressante, d'autant plus que le fait de pouvoir gérer l'échange que nous faisons avec nos proches se prête bien à une application qui permet de gérer ses finances.

### 8.8.1 Vue globale (liste)



GUI 14 - Liste des dettes

#### 8.8.1.1 Composition

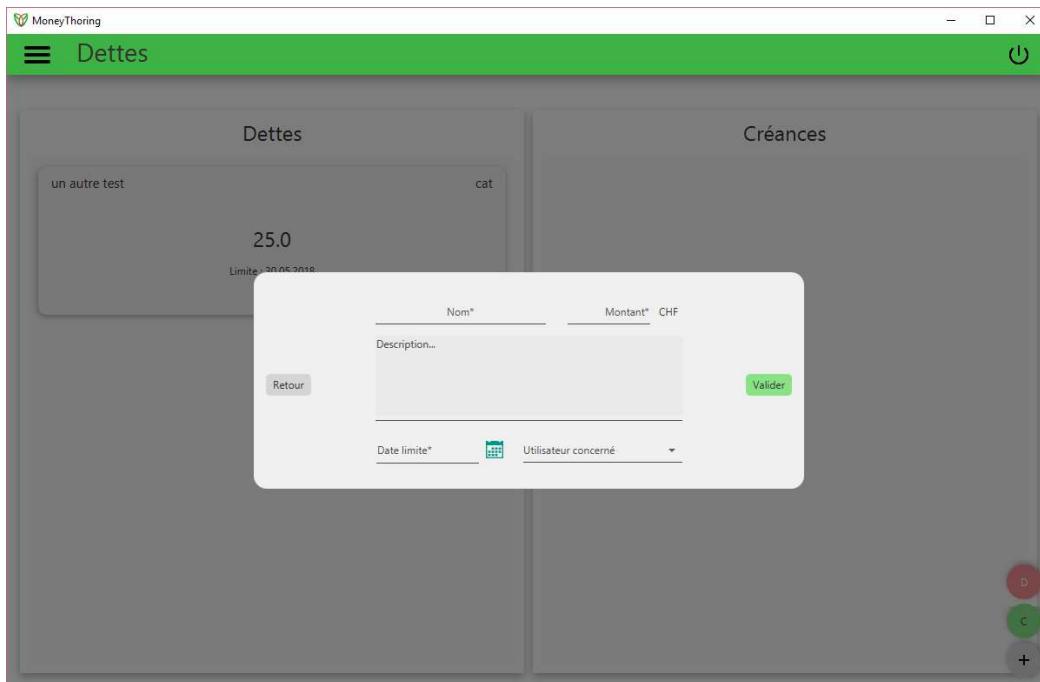
Nous avons divisé la vue en deux parties : à gauche les dettes (donc celles que l'on doit rembourser) et à droite les créances (donc une dette dont on attend le remboursement).

Dans l'élément racine, nous avons un *GridPane* qui remplit ce dernier pour diviser au mieux la vue en deux. Dans chacune des cellules du *GridPane* nous avons inséré un *BorderPane*. Au centre de ce dernier, nous avons mis un *ScrollPane* contenant une *VBox*. De cette façon nous pouvons afficher un certain nombre de dettes sans avoir trop de soucis d'affichage. Nous trouvons au premier plan une *JFXNodeList* afin d'accueillir les boutons qui permettent d'ajouter soit une dette soit une créance, ainsi qu'un *AnchorPane* qui est utilisé pour accueillir la vue du formulaire.

#### 8.8.1.2 Contrôleur

En ce qui concerne l'objet créé pour afficher une dette, nous avons simplement pris une *VBox* dans laquelle nous insérons des labels ainsi qu'un bouton de validation. Ce dernier n'est disponible uniquement pour le créateur de la dette (peu importe qu'il soit le créancier ou le débiteur). D'autres options disponibles uniquement pour le créateur sont la modification et la suppression (annulation) d'une dette. La validation d'une dette va automatiquement créer une dépense chez le débiteur, et un revenu chez le créancier.

### 8.8.2 Formulaire des dettes



GUI 15 - Formulaire de la dette

#### 8.8.2.1 Composition

Les dettes sont assez triviales : nous demandons un nom, un montant et une date limite pour la régler. Il est cependant possible de saisir une description. Si l'autre personne concernée (débiteur ou créancier) est inscrite dans l'application, on peut lier notre dette à son compte. Cela permettra de lister chez l'autre parti ladite dette afin qu'il/elle ne l'oublie pas.

#### 8.8.2.2 Contrôleur

Le contrôleur de cette vue implémente simplement les mécanismes de création, de modification et de suppression que nous avons pu voir précédemment à plusieurs reprises.

## 8.9 Difficultés rencontrées

### 8.9.1 Utilisation de la librairie JFoenix sous IntelliJ IDEA

Le SceneBuilder proposé par IntelliJ IDEA ne permet pas de choisir et de représenter des éléments d'autres librairies graphiques que celle de JavaFX de base.

- Résolution : Utilisation du SceneBuilder proposé par Gluon  
<http://gluonhq.com/products/scene-builder/>

### 8.9.2 Intégration du contrôleur du menu latéral

- Résolution : Pour pouvoir utiliser un contrôleur dans un modèle FXML externe, il faut déclarer un FXMLLoader et lui indiquer le fichier à prendre en compte. Nous devons également déclarer une instance de la classe du contrôleur requis et l'attacher au loader par le biais de la fonction « setController ». Et enfin il faut charger le contenu du FXMLLoader dans un conteneur (de préférence le conteneur à la racine du fichier FXML).

### 8.9.3 Transparence du menu latéral

Pour que le menu latéral s'affiche correctement indépendamment des dimensions de la fenêtre, il a fallu lui attribuer la propriété "fit to parent" qui lui permet d'adapter sa taille par rapport à son conteneur. Cet attribut lui fait donc occuper toute la place du conteneur. Et comme il se trouve au premier plan, les objets qui sont derrière lui (notamment le ContentPane qui reçoit le contenu principal du programme) ne sont pas cliquables, et donc impossible d'effectuer les actions de chaque vue.

- Résolution : Il existe une propriété « mouseTransparent » qui permet de rendre un élément "transparent" aux yeux de la souris, c'est-à-dire que tous les événements de la souris passeront à travers cet élément. Il suffit d'activer et de désactiver cette fonctionnalité, respectivement à la fermeture et à l'ouverture du menu latéral.

### 8.9.4 Conséquences de la migration du projet en projet Maven

Lors de la conversion du projet en projet Maven, la configuration du projet s'est automatiquement mise en Java 5.

- Résolution : Nous avons dû modifier le fichier de configuration du projet pour utiliser Java 9 et faire relayer l'information dans l'équipe.

L'organisation du projet a été modifiée. L'utilisation de Hibernate nécessitait une certaine arborescence qui mettait en place un dossier Ressources. Le code de la GUI utilise une méthode « getResource » qui allait justement chercher les fichiers FXML et CSS dans le dossier des ressources du projet.

- Résolution : Nous avons donc simplement adapté l'organisation des fichiers de la GUI pour que le programme refonctionne.

### 8.9.5 Transactions entre les fenêtres de liste et de formulaire

Nous arrivions à appeler la fenêtre de formulaire, mais nous ne pouvions pas récupérer l'information créée à la confirmation du formulaire.

- Résolution : Nous avons simplement chargé la vue du formulaire (tel que nous avions prévu de le faire à la base) et nous transmettons une référence du contrôleur appelant au formulaire appelé. Le contrôleur qui appelle le formulaire utilise l'interface « IController ». Le formulaire peut donc utiliser cette dernière, afin de transmettre au contrôleur appelant le résultat de la saisie (création ou modification).

### 8.9.6 Modification/suppression d'objets

Avec notre implémentation, nous ne pouvions que créer des objets via notre formulaire.

- Résolution : Après une petite adaptation de l'interface « IController » (ajout de méthodes « deleteItem » et « modifyItem »), il nous est maintenant possible de les modifier et de les supprimer.

### 8.9.7 Déconnexion

La déconnexion de l'utilisateur a été un problème plutôt conséquent en ce qui concerne la gestion des fenêtres. Comme nous n'avions pas tout de suite pensé à l'option de la déconnexion, il a fallu remanier les relations entre la fenêtre de connexion et la fenêtre principale. Précédemment, nous fermions la fenêtre de connexion après avoir créé et affiché la fenêtre principale et nous n'avions donc aucun moyen d'afficher à nouveau la fenêtre de connexion.

- Résolution : Nous avons implémenté un « mini » médiateur *windowManager* qui gère les interactions entre nos deux fenêtres. Nous fournissons la référence vers les fenêtres lors de leur initialisation (qui est automatiquement appelé tout de suite après le constructeur). Nous avons également créé une interface « *IWindow* » qui demande une implémentation des méthodes « *hide* » et « *show* » ; ces méthodes sont exploitées dans notre médiateur.

La déconnexion implique de vider les propriétés de l'utilisateur (compte bancaire, transaction, ...) stockées par le programme, ainsi que les champs de la fenêtre de connexion.

## 9 Logique métier (BLL)

### 9.1 Introduction

Pour la partie du code concernant la logique métier, nous avons décidé de partir sur une séparation entre les modèles et la vraie logique. Les classes modèles sont principalement utilisées pour le mappage avec la DAL, alors que les classes logiques implémentent vraiment les méthodes qui vont permettre les interactions entre les classes elles-mêmes, les classes des interfaces graphiques et celles de la couche d'accès aux données. Pour faire cette séparation, nous avons défini deux packages, le package **model** (bll.model) et le package **logic** (bll.logic).

Les classes modèles déclarent principalement les attributs en lien avec la base de données, ceux qui doivent donc être mappés, et implémentent les méthodes de modifications (setters) et de récupérations (getters) de ces derniers, ainsi que les deux méthodes qui permettent d'ajouter et de mettre à jour les données dans les bases de données.

Les classes logiques, quant à elles, implémentent la vraie logique métier de l'application, c'est-à-dire les interactions entre elles, les attributs qui les concernent et les méthodes nécessaires à la partie graphique. Les méthodes logiques contiennent aussi la méthode de suppression des éléments dans les bases de données, qui s'assure de supprimer l'objet dans la base ainsi que tous les liens éventuels contenus dans les autres objets.

### 9.2 Classes Authentication, KeyGenerator et Mail

La classe *Authentication* est une classe se trouvant dans le package **logic**. Cette dernière n'a pas de modèle car elle ne s'occupe que de la logique concernant l'enregistrement et la connexion d'un utilisateur. Ses tâches principales sont de vérifier la cohérence des informations fournies avec la base de données lors d'un enregistrement ou d'une connexion, de faire le hachage du mot de passe et de vérifier la clé envoyée par email lors de la connexion d'un utilisateur non vérifié avec celle enregistrée en base de données, permettant ainsi de valider le compte et l'adresse email fournie.

La classe *KeyGenerator*, de la même manière que la classe *Authentication*, ne possède pas de modèle. Elle permet simplement la génération d'une clé aléatoire, que nous envoyons à l'utilisateur lorsqu'il s'enregistre sur la base de données en ligne.

La classe *Mail*, faisant partie du package **smtp**, sert à envoyer un email à un utilisateur donné (nom d'utilisateur et adresse email). Cet envoi contient le code d'activation aléatoire généré avec *KeyGenerator* de l'utilisateur donné.

### 9.3 Classe ClientLogic

*ClientLogic* est la classe « maîtresse », c'est celle qui conserve les données de l'utilisateur connecté et qui nous permet de retrouver tous les liens. Comme il ne peut y avoir qu'un seul client connecté à la fois, nous avons décidé d'implémenter cette classe d'après le modèle du *Singleton*. Nous ne pouvons donc avoir qu'une seule instance de cette classe lors de l'exécution de notre programme. C'est donc lors de la connexion, si celle-ci passe, que nous paramétrons l'instance et que nous récupérons toutes les données liées à l'utilisateur.

## 9.4 Les autres classes logiques

En ce qui concerne les autres classes du package **logic**, elles partagent toutes la même structure. Le but principal de ces classes est de bien gérer les interactions entre les différents objets. Certaines de ces classes, comme *CategoryLogic*, sont moins complexes car elles possèdent moins de liens à gérer. Par exemple dans *CategoryLogic*, nous devons simplement gérer les liens lors de la suppression pour éviter des problèmes de dépendance dans la base de données. Alors que d'autres classes, comme *IOTransactionLogic* ou encore *BudgetLogic*, sont beaucoup plus complexes. Par exemple, pour le cas de *IOTransactionLogic*, lors de la création d'une instance, il fallait mettre à jour tableaux et objets utilisés par la GUI ainsi que le solde du compte lié à la transaction. Lors de la mise à jour d'une instance de cette classe, il fallait vérifier derechef ces tableaux, mettre à jour la somme du compte en banque concerné et ainsi de suite.

Nous pouvons dire que les plus grosses classes au niveau de la complexité sont effectivement *IOTransactionLogic*, du fait de ses nombreux liens à différentes structures, *BudgetLogic*, car nous devions gérer les budgets normaux et les budgets partagés, avec des liens à plusieurs catégories et, dans le cas du deuxième, à plusieurs utilisateurs, ainsi que *DebtLogic*, qui fait un lien entre deux utilisateurs et, lorsqu'une dette est confirmée, nous devions gérer la création d'un transaction pour chacun des utilisateurs concernés.

## 9.5 Mappeurs

### 9.5.1 Définition

Notre projet, utilisant le framework Hibernate pour faciliter les interactions entre la base de données et l'application, nécessitait la création d'objets appelés « mappeurs ». Ces derniers ont été utilisés pour transformer des objets provenant de la base de données (entités DAL) en objets Java, et inversément.

Dans notre cas, ils ont servi à passer des objets de type *IDALClassEntity* (PostgreSQL ou Derby) depuis la BLL vers la DAL, et à l'inverse des objets de type *ClassModel* ou *ClassLogic* depuis la DAL vers la BLL.

Ces transtypages permettent donc la communication et l'échange d'informations entre deux environnements de développement différents. Ainsi l'application est capable de traduire une entrée de table de base de données en une classe Java et inversément.

### 9.5.2 Cas d'utilisation

L'utilisation de ces mappeurs est survenue lors de chaque interaction entre la base de données et l'application Java, autrement dit à chaque fois qu'une requête devait avoir lieu sur l'une des bases de données. Pour illustrer les cas d'utilisation, prenons l'exemple de l'entité *Category*.

Avant chaque nouvelle insertion dans la base de données, un objet de type *CategoryModel* est créé au niveau de la BLL. Celui-ci représente l'entité *Category* au format Java. Pour envoyer cette catégorie en base de données, la BLL doit appeler une méthode spécifique de la DAL qui attend un *CategoryModel* en paramètre et qui retourne une *IDALCategoryEntity*. Cette dernière entité représente la catégorie au format base de données. Une fois cette entité reçue, la BLL doit l'envoyer à la méthode « *addCategory* » d'un objet de type *ICategoryRepository*, qui s'occupe de l'insérer dans la base de données spécifiée (PostgreSQL ou Derby).

---

Pour la modification et la suppression, le principe est le même que pour l'insertion à la différence que l'objet échangé possède déjà un ID et par conséquent qu'il est juste mis à jour ou supprimé en base de données.

Avant chaque récupération de données stockées, la BLL doit appeler une méthode spécifique de la DAL qui récupère une ou plusieurs entrées de la base de données et qui retourne une IDALCategoryEntity. Celle-ci représente l'entité Category au format base de données. Pour intégrer cette catégorie, la BLL doit appeler une autre méthode de la DAL qui attend une IDALCategoryEntity en paramètre et qui retourne un CategoryLogic (héritant de CategoryModel). Ce dernier représentant l'entité Category au format Java, la BLL peut donc interagir avec et l'intégrer à l'application.

## 9.6 Difficultés rencontrées

La grosse difficulté au niveau de la couche métier était de bien gérer tous les liens entre les différentes classes, car avec les différentes actions (suppression, mise à jour, création), un changement était vite oublié.

Une autre difficulté était de bien gérer les interactions entre la DAL et la GUI. Cela a mené à beaucoup de fonctions particulières que nous aurions pu éviter avec une autre modélisation de nos interactions avec la base de données. En effet, dans notre conception, tout ce qui va être mis dans la base de données a un équivalent d'instance dans la couche métier, même quand cela ne concerne pas l'utilisateur connecté (dettes partagées, budgets partagés). Alors que nous ne devrions pas avoir d'objets concernant d'autres utilisateurs que l'utilisateur connecté. Dans ces cas-là, nous aurions dû travailler uniquement avec des IDs et des requêtes précises, mais l'implémentation d'Hibernate et les nombreux problèmes que nous avons eu avec nous ont fait partir dans une version plus "simple" pour nous.

# 10 Tests

## 10.1 Liste des tests effectués

Entité à tester	Action	Test réalisé	Résultat attendu		Résultat Derby		Problème
			Résultat PostgreSQL	Résultat Derby	Résultat PostgreSQL	Résultat Derby	
Enregistrement	Saisie avec des champs vides	-	●	●	●	●	
	Saisie avec un nom d'utilisateur existant	-	●	●	●	●	
	Saisie avec un email existant	-	●	●	●	●	
	Saisie avec un email incorrect	-	●	●	●	●	
	Saisie avec deux mots de passe différents	-	●	●	●	●	
	Saisie avec des informations correctes	-	●	●	●	●	
Connexion	Réception du mail avec le code d'activation à l'enregistrement	-	●	●	●	●	
	Connexion avec des champs vides	-	●	●	●	●	
	Connexion avec un nom d'utilisateur inexistant	-	●	●	●	●	Message d'erreur toujours présent si déconnexion du client
	Connexion avec un email inexistant	-	●	●	●	●	Message d'erreur toujours présent si déconnexion du client
	Connexion avec un nom d'utilisateur existant et un mot de passe incorrect	-	●	●	●	●	Message d'erreur toujours présent si déconnexion du client
	Connexion avec un email existant et un mot de passe incorrect	-	●	●	●	●	Message d'erreur toujours présent si déconnexion du client
Activation	Connexion avec un nom d'utilisateur et un mot de passe existants	-	●	●	●	●	
	Connexion avec un email et un mot de passe existants	-	●	●	●	●	
	Accès à la zone d'activation à la connexion d'un compte non activé	-	●	●	●	●	
	Accès au tableau de bord à la connexion d'un compte activé	-	●	●	●	●	
Déconnexion	Saisie d'un code d'activation incorrect	-	●	●	●	●	
	Réception du mail avec le code d'activation à la demande de renvoi du code	-	●	●	●	●	
	Accès au tableau de bord et activation du compte à la saisie du bon code	-	●	●	●	●	
Suppression	Accès à la zone de connexion à la déconnexion du client	-	●	●	●	●	Champs de la partie connexion plus accessibles avec le curseur
	Connexion avec client supprimé	-	●	●	●	●	Champs de la partie connexion plus accessibles avec le curseur
							Client pas supprimé si dépendances avec autres tables (conflict DB/BLL)

		Compte bancaire										Transaction																			
		Création					Modification					Suppression					Création					Modification					Suppression				
Catégorie	Attribut	Choix de la couleur d'une catégorie	Création avec des champs vides	Création d'une catégorie	Modification avec des champs vides	Modification d'une catégorie avec changement de couleur	Modification d'une catégorie avec changement de nom	Suppression d'une catégorie quelconque	Suppression de la catégorie par défaut	La suppression d'une catégorie quelconque, remplace la catégorie des transactions concernées par celle par défaut du client	Création avec des champs vides	Création d'un compte bancaire	La création d'un deuxième compte bancaire par défaut, retire l'attribut "par défaut" du premier compte	Tableau de bord	Le résumé du compte par défaut s'affiche sur le tableau de bord Si aucun compte par défaut n'existe, le tableau de bord ne se remplit pas	Modification avec des champs vides	Modification d'un compte bancaire	La modification d'un compte bancaire en compte par défaut, retire l'attribut "par défaut" du compte par défaut si il existe	Création avec des champs vides	Création d'un revenu	Création d'une dépense	La création d'un revenu négatif le convertit en positif	Modification avec des champs vides	Modification d'un revenu	Modification d'une dépense	La modification d'un revenu en négatif le convertit en positif	Suppression d'un revenu	Suppression d'une dépense			
		Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré	Pas géré						

		Budget																			
		Création avec des champs vides	La création d'un budget avec une valeur de plafond négative la convertit en positif	Création avec une date de fin inférieure à la date de début	Création d'un budget	Modification avec des champs vides	La modification d'un budget avec une valeur de plafond négative la convertit en positif	Modification avec une date de fin inférieure à la date de début	Modification d'un budget	Création d'un budget avec plusieurs catégories	Modification des catégories d'un budget	Suppression des catégories d'un budget	Création d'un budget partagé avec plusieurs participants	Modification des participants d'un budget partagé	Suppression des participants d'un budget partagé	Suppression d'un budget normal	La suppression d'un budget partagé avec participants, par son créateur, implique le changement de possession du budget par un autre membre	La suppression d'un budget partagé avec participants, par un membre, ne supprime pas le budget mais seulement son lien avec le membre			
Création		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
	Modification	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
Catégories du budget		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
	Clients du budget partagé	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
Suppression		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			

		Dette																					
		Création					Modification					Suppression					Validation						
		Création d'une dette avec des champs vides	Création d'une dette avec un ou plusieurs champs obligatoires manquants	Création d'une dette contenant uniquement les champs obligatoires	Création d'une dette complète	Création d'une créance avec des champs vides	Création d'une créance avec un ou plusieurs champs obligatoires manquants	Création d'une créance contenant uniquement les champs obligatoires	Création d'une créance complète	Modification d'une dette avec des champs vides	Modification d'une dette avec un ou plusieurs champs obligatoires manquants	Modification d'une dette contenant uniquement les champs obligatoires	Modification d'une dette complète	Modification d'une créance avec des champs vides	Modification d'une créance avec un ou plusieurs champs obligatoires manquants	Modification d'une créance contenant uniquement les champs obligatoires	Modification d'une créance complète	Suppression d'une dette	Suppression d'une créance	Validation d'une dette	Validation d'une dette partagée	Validation d'une créance	Validation d'une créance partagée
<b>Création</b>	Date pas mise en rouge	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	Date pas mise en rouge	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
<b>Modification</b>	Date pas mise en rouge	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	Date pas mise en rouge	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
<b>Suppression</b>	Date pas mise en rouge	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	Date pas mise en rouge	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
<b>Validation</b>	Validation d'une dette	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	Validation d'une dette partagée	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

## 10.2 Bugs restants

Sur l'ensemble des tests réalisés, seule une petite partie ne passe pas ou génère parfois des erreurs. Ces dernières sont divisées en 2 catégories : des détails qui auraient dû être gérés par la GUI et un bug.

Les cas non gérés de la GUI ne représentent en soit pas une source d'erreur au niveau de l'application. En effet, cette dernière ne plante pas pour des problèmes d'affichage de messages d'erreur ou de champs inaccessibles.

En revanche notre plus gros problème est la suppression d'un client, que ce soit en ligne ou hors ligne avec Derby. Dans le cas le plus simple, autrement dit un client sans compte bancaire ni autres données enregistrées, la suppression fonctionne. Cependant, dès que celui-ci crée des entrées dans la base de données, liées à d'autres tables, l'application ne plante pas, mais ne fait tout simplement pas ce qu'on lui a demandé.

Cette erreur provient probablement du fait que nous avons géré à double les répercussions d'actions telle que la suppression. En créant le script de base de données, nous avons créé un code tel que nous avons appris à le faire, avec des « ON DELETE ..., ON UPDATE » sur les clés étrangères des tables de données. Ceci était totalement justifié et mûrement réfléchi puisque nous voulions que la modification et suppression d'une donnée se répercute sur les autres tables.

Dans un deuxième temps, quand nous avons commencé à coder l'application, nous nous sommes rendu compte que la suppression d'une catégorie ne devait pas engendrer la suppression des transactions associées (dont la clé étrangère de catégorie ne doit pas être nulle). De ce fait nous avons mis en place un trigger qui, avant la suppression d'une catégorie, modifie la catégorie de la transaction associée par la catégorie par défaut de l'utilisateur (qui elle n'est jamais supprimée).

Ces deux choses mises en place, lors de la suppression d'un client, le trigger tente de remplacer chaque catégorie de transaction par celle par défaut, qui a déjà été supprimée. Une erreur étant provoquée, la base de données stoppe le processus de suppression et le client reste donc intacte.

## 11 Conclusion

### 11.1 Déroulement du projet

Globalement, le projet s'est déroulé sans trop de problèmes. Sa mise en route s'est faite gentiment, c'était la première fois que nous travaillions tous ensemble, nous avons dû nous organiser, nous assigner des tâches et nous mettre d'accord sur les objectifs et l'application à mettre en place.

Une fois la répartition faite, l'équipe s'occupant de l'interface graphique s'est directement attelée à sa tâche et avançait de manière régulière. En ce qui concerne l'équipe des bases de données et de leurs accès, elle a très rapidement rencontré des problèmes, comme le stockage de la base de données distante, l'utilisation du framework Hibernate ou encore des problèmes avec la base de données locale. Ce sont malheureusement ces problèmes qui, de part leur accumulation et leur temps de résolution, ont énormément ralenti le projet.

En effet, ces problèmes nous empêchaient de tester les communications entre l'interface graphique, la couche métier et les bases de données, l'équipe travaillait à l'aveugle sans pouvoir tester les interactions. De plus, nous avons traîné ces problèmes jusqu'à la moitié du temps que nous avions à disposition.

Néanmoins, une fois ces problèmes résolus, nous avons mis les bouchées doubles et fait fonctionner plusieurs fonctionnalités assez rapidement. À partir de là, l'avancement du projet s'est accéléré et nous commençons enfin à avoir des résultats, ce qui fut excellent pour le moral général de l'équipe.

À ce stade du projet, et bien que l'avancement ait fait un bond, nous sommes restés现实ists et savions que nous n'aurions pas le temps de tout terminer, surtout que jusque là nous avions uniquement implémenté la partie avec la base de données distante dans les couches métier et graphique. Nous nous sommes donc concertés et nous avons pris la décision d'abandonner la synchronisation entre les deux bases de données. En effet, nous n'étions même pas sûrs de pouvoir implémenter l'accès à la base de données locale en parallèle à celle déjà en place, donc la synchronisation entre les deux était clairement infaisable.

Pour donner suite à cette décision, nous avons continué notre bonne avancée sur le projet. Nous avons, tout en continuant à implémenter nos fonctionnalités avec la base distante, mis en place la base de données local et adapté le code nécessaire pour faire fonctionner les deux en parallèle. Une fois toutes nos fonctionnalités implémentées, nous avons arrêté l'implémentation du code pour nous concentrer sur les tests et les rapports.

Durant la phase de tests, nous avons soulevé quelques problèmes que nous avons pu corriger, les corrections n'étant pas excessives et ne mettant pas en danger le bon fonctionnement de l'application, et d'autres que nous avons dû nous résoudre à laisser tel quel pour éviter les mauvaises surprises. Il n'est jamais très bon de faire de grosses modifications au dernier moment avec peu de temps pour les rattraper si besoin, nous avons donc décidé de ne pas y toucher.

Pour en finir avec ce déroulement, nous trouvons qu'il s'est bien passé. Il est normal d'avoir rencontré des difficultés et nous estimons les avoir bien gérées. De plus, si nous avons dû faire l'impasse sur quelques fonctionnalités, ce n'est pas uniquement dû au retard mais aussi à une mauvaise estimation de notre part quant à la charge de travail. En effet, c'était la première fois que nous réalisions un tel projet et, comme on nous avait prévenu, nous avons effectivement mal géré les estimations. Nous avons réalisé par la suite que notre projet était

bien plus complexe que ce que nous imaginions au début et qu'il était trop gros pour être réalisé dans son entièreté avec une limite de temps aussi brève, surtout au vu de nos compétences sur certaines technologies au début du projet. Mais c'est un très bon enseignement et nous saurons sans aucun doute beaucoup mieux estimer les projets futurs. De plus, d'avoir réalisé cela nous rend d'autant plus fiers du résultat obtenu et de l'expérience acquise.

## 11.2 Fonctionnement du groupe

En tant que chef de projet, c'est moi qui vais m'occuper de spécifier cette partie du rapport.

Tout d'abord, au niveau de la répartition des tâches, Bryan Curchod et François Burgener s'occupaient exclusivement de la partie interface et graphique (GUI). Hélène Reymond et Guillaume Zaretti, quant à eux, s'occupaient des parties concernant les bases de données et la couche d'accès aux données (DB et DAL). Quant à moi, je me suis occupé principalement de la logique métier de l'application (BLL) ainsi que des tâches qu'incombent au chef de projet.

Nous avons décidé en équipe de la répartition du groupe. Comme JavaFX (GUI) n'était connu de personne, nous avons décidé de mettre deux personnes dessus, de telle sorte qu'elles puissent s'entraider si besoin. Ensuite, concernant la DB et la DAL, nous avons décidé de mettre deux personnes également pour permettre une mise en place rapide de la base de l'application. Une des deux devait, une fois la base implémentée, rejoindre la BLL. Pour cette dernière, comme c'était la partie connue et donc celle avec la plus petite charge de travail, elle m'a été assignée. De cette façon, je pouvais prendre du temps pour observer et suivre l'avancée des autres membres et les aider si besoin.

L'équipe GUI a bossé de manière très indépendante dès le début, ils savaient quoi faire et n'avaient pas besoin d'aide externe. Dès qu'ils avaient besoin d'une méthode de la BLL, ils faisaient la demande nécessaire et continuaient à avancer sans trop de problèmes.

L'équipe DB et DAL savait aussi ce qu'elle devait faire. Hélène s'est occupée de la partie DB pendant que Guillaume s'occupait de la partie DAL. L'équipe a eu de gros problèmes dû aux technologies choisies et a eu de la peine à les résoudre rapidement. Néanmoins et malgré les problèmes, ils continuaient d'avancer. Ce qui eut pour résultat, une fois les problèmes résolus, que tout leur travail fonctionnait presque intégralement et qu'il ne leur restait plus que des détails.

L'équipe BLL, au début, n'a pas beaucoup avancé. En effet, étant seul et l'équipe DB et DAL ayant des problèmes importants, j'ai mis une bonne partie de mes efforts pour les aider, tout en faisant en parallèle la modélisation de la couche métier. Malgré ces péripéties, j'ai avancé de manière régulière, avec un léger retard sur les besoins de la GUI. Une fois les bases de données fonctionnelles et la couche d'accès à celles-ci opérationnelle, Hélène est venue compléter l'équipe BLL, laissant ainsi Guillaume s'occuper des derniers détails de la DAL et des bases de données.

De manière globale, l'équipe a, avec cette disposition, travaillé de manière efficace durant toute la durée du projet. L'entente au sein du groupe, bien que pas toujours optimale, est restée à un bon niveau entre la plupart des membres.

Je vais maintenant faire une brève description de quelques problèmes au niveau du travail fourni par les équipes et leurs membres.

Concernant l'équipe GUI, j'ai trouvé qu'il y avait un manque de rigueur et de méthodologie. Cela s'est surtout remarqué sur les détails, mais les détails sont parfois importants. En effet, lors de la phase de tests, pleins de petits détails, comme les vérifications de saisie aléatoires ou des incohérences graphiques entre certaines pages, sont ressortis. Ce ne sont pas des problèmes qui font planter l'application, certes, mais ce genre de détails peut entacher l'image d'un produit et de son équipe. Nous-mêmes étant des consommateurs, nous savons très bien à quel point l'utilisateur peut être pointilleux et nous devons y prêter attention. A part cela, l'équipe GUI ne m'a posé aucun autre problème. Ils étaient toujours à l'écoute et effectuaient leur travail sans souci, même les tâches fastidieuses comme le maintien du journal de travail et du rapport.

Concernant l'équipe BLL, j'ai commencé de manière très méthodologique et rigoureuse, je commentais tout ce que je codais et mon journal de travail était toujours à jour. Malheureusement, cela s'est dégradé au fil du temps. Plus le projet avançait, plus il y avait de liens à gérer en BLL et de demandes émanant de la GUI. Étant seul, j'avais beaucoup de mal à tenir le rythme avec la rigueur que je m'étais jusque là imposée. J'ai donc commencé à laisser de côté les commentaires du code pour répondre efficacement à la charge de travail, ce qui n'est pas forcément une excellente idée. Les commentaires et le rapport sont toujours plus faciles à réaliser au fur et à mesure que d'un seul coup à la fin. Une fois Hélène dans l'équipe, ma charge de travail a diminué et j'ai pu récupérer une partie de la rigueur perdue, bien qu'étant à la fin du projet, nous devions nous concentrer sur le fonctionnement des fonctionnalités.

Concernant l'équipe DB et DAL, j'ai aussi remarqué un manque certain de rigueur et de sérieux du coté de Guillaume pour tout ce qui est organisation et maintien du journal de travail. Au niveau du code, il était toujours très efficace, mais dès qu'il sortait des aspects techniques, comme pour commenter le code ou maintenir son journal, impossible d'avoir un retour régulier. Cela se remarque quand on compare les journaux de travail de l'équipe. De plus, il y avait un manque de participation aux discussions les deux dernières semaines, surtout lorsqu'il fallait se rencontrer en dehors des cours (que ce soit par chat ou en personne), comme s'il esquivait la chose. Or cela est inadmissible lorsqu'on travaille en équipe. Hélène, en revanche, ne m'a pas posé de problèmes. Son journal de travail était toujours à jour, son code et ses commentaires également, et elle poussait même les autres à le faire pour ne pas se retrouver avec une charge de travail finale trop grande. Elle était visiblement bien plus consciente que les autres de la charge de travail lorsqu'on laisse ce genre de chose pour la fin. En plus de cela, elle s'est occupée de maintenir le rapport du projet à jour avec les différentes parties de chaque personne, ce qui nous a évité un énorme travail de dernière minute.

En tant que chef de projet, je pense avoir été un bon leader, à l'écoute des autres membres, et avoir réussi ma mission. J'ai fait en sorte que les choses soient bien faites et que tous les membres participent et fournissent le travail demandé. J'ai peut-être parfois été un peu trop pointilleux et exigeant pour des détails que d'autres membres du groupe trouvaient sans importance, mais je pense avoir globalement su gérer mon rôle.

Malgré ces points négatifs, l'équipe a tout de même réussi à s'entendre, à travailler ensemble malgré les différences d'opinions et à réaliser le travail demandé. De plus, ce fût une bonne expérience que d'être confronté à des personnes très différentes au sein du groupe, cela nous enseigne à nous adapter aux autres et à communiquer pour faire fonctionner les choses. Malgré quelques prises de têtes, je suis très satisfait de la cohésion de l'équipe, de sa bonne ambiance globale et de l'effort fourni par tous ses membres.

## 11.3 Avis personnel des membres du groupe

### 11.3.1 Bryan Curchod

Le projet arrivant à sa fin, il est temps d'en faire le bilan. Le développement de l'interface graphique a été un défi intéressant. JavaFX est une technologie intéressante, particulièrement grâce au fait de pouvoir séparer la maquette graphique d'un contrôleur. Je regrette peut-être de ne pas avoir pu découvrir toutes les possibilités de cette technologie. En ce qui concerne le déroulement du projet, il y a eu beaucoup de haut et de bas (comme dans beaucoup de projets), mais je pense qu'il est important de noter deux "bas". Sur les dernières semaines, une certaine tension pouvait se faire sentir à cause d'un membre du groupe. Afin de ne pas m'éterniser sur le sujet, je dirais simplement qu'il y avait un cruel manque de communication et peut-être d'investissement de sa part. Le second point négatif est le retard que l'on a pris durant le projet : la partie base de données a mis un certain temps à se mettre en place, et ce retard s'est fait ressentir sur les autres "couches" du projet (DAL, BLL, GUI... bien qu'avec un peu de recul je me rends compte que la GUI aurait pu progresser indépendamment) et nous avons dû mettre de côté quelques fonctionnalités. Malgré ces impairs, je pense que le projet se termine sur une note positive.

### 11.3.2 Daniel Gonzalez Lopez

J'ai trouvé ce projet fort intéressant. C'est la première fois que je faisais un projet d'une telle ampleur et la diversité des compétences requises ainsi que leur maîtrise est extrêmement rafraîchissante. En effet, on apprend beaucoup de chose lors de tels projets.

Tout d'abord, et c'est pour moi le point essentiel d'un tel travail, vient l'organisation et la cohésion de l'équipe. Il est déjà souvent difficile de s'organiser correctement seul dans des projets, mais lorsque cela doit se faire en groupe, cela rend la tâche bien plus complexe et intéressante. En tant que chef de projet, je m'étais ajouté une certaine pression pour faire en sorte que tout se passe bien, que l'équipe soit efficace et que tout le monde participe aux discussions. Ce ne fut pas facile, et cela du début à la fin. Certains membres sont toujours un peu réticents au moment de travailler d'une manière qui n'est pas dans leurs habitudes, alors que d'autres sont plus explosifs et partent rapidement dans tous les sens. J'ai essayé de faire comprendre l'importance de la rigueur dans certaines tâches, pour éviter de repasser 50 fois dans le même fichier, mais c'est visiblement quelque chose qui a de la peine à venir naturellement certaines personnes. Cependant, je pense avoir globalement réussi à garder une équipe efficace et rigoureuse.

En ce qui concerne le projet en soit, ce dernier a été plutôt bien géré. Malgré un retard évident dû à certaines nouvelles technologies, nous avons réussi à réaliser un projet complet et fonctionnel. Nous avons bien évidemment dû faire l'impasse sur quelques points, mais une bonne partie a tout de même été réalisée et je suis personnellement très fier du résultat que nous avons obtenu.

Je suis aussi très satisfait de l'équipe, qui s'est globalement toujours valorisée par sa disponibilité et sa motivation. Hélène Reymond et Bryan Curchod sont, je pense, de très bons éléments sur lesquels on peut compter sans souci et qui travaillent de manière efficace. Hélène a tendance à pousser les gens à travailler, car elle n'aime pas faire les choses à la dernière minute et c'est une excellente qualité. Bryan quant à lui est un peu plus « relaxe », mais dès qu'il faut s'y mettre, les choses sont généralement bien faites. J'ai eu plus de peine avec François Burgener

et ce surtout sur la fin du projet, qui dès que je lui faisais une demande, était réticent et c'était toujours des réponses négatives qui sortaient en premier. Fondamentalement, il finit toujours par fournir le travail demandé, mais il m'a fallu le pousser un peu plus que les deux précédents. Passons maintenant à Guillaume Zaretti... En ce qui concerne ses capacités, il est extrêmement doué pour les aspects techniques, notamment dans tout ce qui concerne l'architecture des logiciels. Sans lui, nous n'aurions probablement pas utilisé le framework Hibernate, qui nous a certes mis en retard, mais qui nous a quand même fait gagner un peu de temps vers la fin. Sans ce framework, je ne pense pas que nous aurions pu intégrer deux bases de données différentes (distante et locale) dans notre logiciel. Cependant, il n'excelle de loin pas en travail de groupe. J'ai eu plusieurs heures de perdues à expliquer et réexpliquer certains principes lorsqu'on travaille en groupe, par exemple le fait de répondre aux messages lorsqu'on essaie d'organiser une séance, ou encore ne pas partir dans tous les sens et tester des dizaines de choses pour finir par repasser dessus parce que ça avait été mal fait. C'est très certainement avec lui que j'ai eu le plus de mal, surtout pour lui faire tenir à jour son journal de travail régulièrement.

Globalement, j'ai eu une très bonne expérience avec l'équipe et ce projet. Mais tout n'était pas rose. En effet, je suis assez insatisfait car, quand bien même notre application tourne bien et est assez complète, il y a pleins de petits détails pour lesquels nous n'avons pas été assez rigoureux, principalement par manque de temps, et qui pourraient être beaucoup mieux. Pour ma part, c'est principalement la gestion des exceptions qui m'embête le plus. Dans mes méthodes, j'ai principalement juste affiché les exceptions (pour le débogage) alors qu'il y a clairement moyen de mieux les utiliser, mais je n'ai malheureusement pas eu le temps de m'y pencher. Pareil pour certaines modifications vers la fin du projet, en plein stress pour finir certaines fonctionnalités, on pense moins à éviter les duplications de code et autres détails du genre. Je suis également déçu parce que je pense que nous aurions pu mieux réussir et prendre moins de retard, mais ce genre de projet est là aussi pour nous enseigner que tout ne va pas toujours pour le mieux.

Ce projet aura donc été très instructif et riche en expériences pour la suite. Et j'ai hâte des projets futurs.

### 11.3.3 François Burgener

Un début assez difficile, car il nous fallait tout mettre en place et apprendre les nouvelles technologies. Pour ma part j'ai dû mettre en place l'interface graphique de l'application avec JavaFX. Cela était un assez grand défi, car ne connaissant pas grand-chose en GUI, j'ai dû prendre du temps pour m'adapter et comprendre comment cela fonctionnait. Nous avons eu pas mal de problèmes qui étaient liés à l'interaction des différentes fenêtres. J'aurais bien aimé pouvoir l'exploiter plus en profondeur, car JavaFX est un outil assez puissant.

Pour ma part je trouve que nous avons pris trop de temps pour faire fonctionner les premières fonctionnalités de l'application. Mais après ça nous sommes allés relativement vite pour l'implémentation des différentes fonctionnalités.

Sur ces dernières semaines de projet, il y a eu certains conflits avec un des membres du groupe. Dès qu'on essayait de s'organiser pour travailler en groupe il ne nous répondait pas et donc il n'était pas présent quand on avait besoin de lui.

En général, ce projet s'est relativement bien passé. Nous avons pu faire fonctionner la majorité des fonctionnalités demandées.

### 11.3.4 Guillaume Zaretti

J'ai eu du plaisir à travailler en équipe et à organiser un projet, dont les tâches ont été bien attribuées. Le fait d'avoir organisé un découpage de l'application a permis à l'équipe de travailler de façon efficace en se focalisant uniquement sur certaines zones de code. C'est la première fois que je développe une application en équipe, le résultat escompté est positif. Je ne pensais pas que l'on allait aboutir à une application utilisable avec un design aussi soigné. Le fait d'avoir découpé le développement en 3 parties ne m'a pas facilité le suivi du code écrit dans l'interface graphique, dans la couche supérieure ainsi que certains choix d'implémentation et de mise en œuvre dans les codes appelant mes méthodes.

Dans la mise en œuvre, je n'ai pas choisi d'exploiter la généricité, ce qui aurait pu nous faire gagner du temps en factorisant le code étant donné que chaque repository partage les mêmes méthodes de CRUD et que le code pour les méthodes de base est très ressemblant. Le choix de ne pas exploiter la généricité est induit par le fait que nous avons pris du retard sur le début et que nous utilisions deux bases de données différentes avec un ORM que nous n'avons jamais utilisé.

La durée du projet est trop courte pour pouvoir estimer correctement certaines tâches dans la planification. Le retard accumulé au départ m'a forcé dans certaines décisions par exemple avec JPA ou la généricité, car à ce moment nous n'avions pas encore trouvé un rythme régulier qui puisse nous donner un repère d'estimation pour aider à la décision de certains choix. Donc avec le retard j'ai fini par tomber dans la logique « il faut que ça marche avant tout » au détriment de la qualité dans le code et la structuration.

### 11.3.5 Hélène Line Reymond

De manière générale je dirais que le projet s'est bien déroulé, malgré le retard pris au début à cause des multiples problèmes à résoudre au niveau d'Hibernate. Nous avons implémenté les fonctionnalités les plus conséquentes, que nous avions prévues lors de la rédaction du cahier des charges et notre application est à la hauteur de nos attentes. De ce fait je peux assurer que nous sommes contents et fiers d'en être arrivés jusque-là et d'avoir une application fonctionnelle.

Néanmoins, la gestion de groupe n'a pas toujours été facile. Il est difficile, pour une première fois, de travailler en groupe de 5 personnes. Chacun a un niveau de développement, des connaissances et une manière de travailler différentes des autres. Dans mon cas, cela m'a fait parfois perdre du temps, car je devais repasser derrière une personne qui laissait des erreurs dans son code. J'ai trouvé très difficile d'être confrontée et de travailler avec des personnes qui ont moins de rigueur que moi. Quand je développe ou quand j'écris un rapport, j'ai tendance à me relire plusieurs fois et à tester ce que je fais avant de le transmettre plus loin. Ce qui, selon ma découverte, n'est pas le cas de tout le monde.

C'est pourquoi je me suis proposée pour écrire le rapport et corriger ceux des autres membres du groupe. Ayant un bon niveau d'orthographe/grammaire et sachant être concentrée quand je lis quelque chose, j'ai préféré prendre ce rôle (malheureusement) par manque de confiance dans le reste de mon groupe. Toutefois, cette tâche m'a enlevé un certain plaisir à avancer dans ce projet, car j'ai passé beaucoup de temps à taper du texte plutôt qu'à développer et aider les autres. J'aurais aimé coder plus, mais il fallait bien quelqu'un pour écrire le compte rendu de notre dur labeur.

Malgré ça, j'ai quand même pu apprendre plein de choses durant mes moments de développement. J'ai notamment découvert Hibernate, PostgreSQL et Derby. J'ai pu me

familiariser avec la DAL et la BLL, de ce fait je sais comment sont gérées les interactions entre l'application et la base de données. C'est un très bon outil que j'utiliserais dans le futur si l'occasion se présente.

Pour terminer je tiens à signaler que nous avons rencontrés quelques problèmes avec Guillaume Zaretti. Il est très compétent et sait beaucoup plus de choses que nous dans le domaine technique. C'est d'ailleurs lui qui nous a présenté Hibernate. Mais il a beaucoup de peine à comprendre ce que représente et implique un travail en groupe. Il a eu tendance à se rendre indisponible dans les moments les plus critiques et à ne pas fournir au plus vite ce qu'on lui demandait. Ce genre de cas est problématique, car en se retrouvant à courir après les membres du groupe posant problème, on perd encore une fois beaucoup de temps et d'énergie qu'on pourrait tout simplement investir dans nos propres tâches.

## 11.4 Améliorations possibles

Au niveau des améliorations possibles, il faudrait commencer par revoir les quelques bugs découverts pour que nous puissions repartir d'une implémentation propre. A cela nous pouvons ajouter une meilleure gestion des exceptions, qui est pour l'instant presque inexisteante.

Ensute, une bonne chose serait de repasser dans le code et de le factoriser, car il y a sans doute moyen d'améliorer ce dernier.

Avant de commencer le projet, l'équipe a bien passé du temps à définir ses besoins. Dans la discussion, le souhait de réutiliser l'application et de pouvoir continuer à l'adapter par la suite se faisait ressentir. De ce fait, nous avons dès le départ réfléchi à une application de gestion de budget qui puisse être évolutive et décomposée en plusieurs packages. Pour ce faire nous avons retenu une application développée en couches, afin de rendre possible une future adaptation de l'application en ajoutant et remplaçant certaines parties. Le fait d'avoir développé l'application MoneyThoring en couches permettra par la suite de créer un service web directement côté serveur, afin d'optimiser le trafic entre la base de données centrale et les clients, car le service web avec Hibernate sera au plus proche du serveur où la base de données centralisée est hébergée. D'autant plus que la création d'un service web est nécessaire pour l'interopérabilité et l'intégration de différents types de client, par exemple : client web, client desktop, client mobile, etc... Cela est possible en réutilisant le package de la couche d'accès aux données qui se trouve sur le client actuel et en le mettant côté serveur, il en va de même pour la partie logique. Côté serveur la logique offrira plus de sécurité et une meilleure gestion des règles métier. Donc pour produire une infrastructure de qualité et optimiser les flux de données, nous devons créer des packages réutilisables en limitant le couplage, afin de les intégrer ou les déplacer dans différentes parties d'une future architecture.

En plus de l'expansion du service à d'autres plateformes, une amélioration intéressante pourrait être d'ajouter les récurrences, que nous pensions pouvoir implémenter mais que nous avons dû abandonner par manque de temps.

Ou encore les emails de rappel, par exemple lorsqu'une facture arrive ou que la date d'échéance d'une dette se rapproche.

## 12 Bibliographie / webographie

### 12.1 BLL

#### 12.1.1 Gestion des SQL dates

- <https://stackoverflow.com/questions/10413350/date-conversion-from-string-to-sql-date-in-java-giving-different-output>

#### 12.1.2 JavaMail

- <https://javaee.github.io/javamail/>
- <https://codes-sources.commentcamarche.net/faq/10706-envoi-d-un-mail-avec-javamail>
- [https://www.tutorialspoint.com/java/java\\_sending\\_email.htm](https://www.tutorialspoint.com/java/java_sending_email.htm)

#### 12.1.3 Hash d'un mot de passe

- <https://stackoverflow.com/questions/2860943/how-can-i-hash-a-password-in-java>
- <https://howtodoinjava.com/security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/>

#### 12.1.4 Regex de validation d'emails

- <https://stackoverflow.com/questions/624581/what-is-the-best-java-email-address-validation-method>

### 12.2 DAL

#### 12.2.1 Requêtes avec Hibernate

- <https://www.mkyong.com/hibernate/hibernate-query-examples-hql/>
- <https://examples.javacodegeeks.com/enterprise-java/hibernate/hibernate-query-language-example/>
- <http://www.codejava.net/frameworks/hibernate/hibernate-query-language-hql-example>

### 12.3 DB

#### 12.3.1 Création des scripts et requêtes

- <https://stackoverflow.com/>
- <https://db.apache.org/derby/manuals/>
- <https://www.postgresql.org/docs/>

## 12.4 GUI

### 12.4.1 JavaFx

- <https://openclassrooms.com/courses/les-applications-web-avec-javafx>
- <https://docs.oracle.com/javase/8/javafx/api/toc.htm>

### 12.4.2 Jfoenix

- <http://www.jfoenix.com>
- [https://www.youtube.com/watch?v=22QlOj6JVe4&list=PLhs1urmduZ29LNYi\\_MaoU60Je\\_mQ6Aei6A](https://www.youtube.com/watch?v=22QlOj6JVe4&list=PLhs1urmduZ29LNYi_MaoU60Je_mQ6Aei6A)

### 12.4.3 Icônes

- <https://material.io/tools/icons/?style=baseline>

## 13 Table des illustrations

### 13.1 Architecture

Architecture 1 - Schéma général de l'architecture de MoneyThoring .....	8
-------------------------------------------------------------------------	---

### 13.2 Bases de données (DB)

DB 1 - Schéma relationnel de la base de données .....	10
-------------------------------------------------------	----

### 13.3 Couche d'accès aux données (DAL)

DAL 1 - Importation du schéma de base de données PostgreSQL .....	14
DAL 2 - Arborescence de projet après la configuration DB de PostgreSQL .....	14
DAL 3 - Code de test de l'intégration d'Hibernate .....	15
DAL 4 - Résultat de l'exécution du premier test d'Hibernate .....	15
DAL 5 - Dépendances Maven ajoutées au pom.xml .....	16
DAL 6 - Configurations d'Hibernate manquantes .....	16
DAL 7 - Résultat de l'exécution du troisième test d'Hibernate .....	16
DAL 8 - Dépendance Maven ajoutée au pom.xml .....	17
DAL 9 - Mise à jour du driver utilisé pour PostgreSQL .....	17
DAL 10 - pom.xml .....	17
DAL 11 - hibernate.pgsqI.cfg.xml .....	18
DAL 12 - Structure de la couche d'accès aux données .....	18

### 13.4 Interface graphique (GUI)

GUI 1 - Fenêtre de connexion à MoneyThoring .....	21
GUI 2 - Fenêtre principale, menu ouvert .....	22
GUI 3 - Fenêtre des comptes bancaires .....	23
GUI 4 - Vue détaillée d'un compte bancaire .....	24
GUI 5 - Formulaire des comptes bancaires .....	25
GUI 6 - Liste des catégories .....	26
GUI 7 - Formulaire des catégories (modification) .....	27
GUI 8 - Aperçu de la liste des transactions .....	27
GUI 9 - Formulaire des transactions (création) .....	28
GUI 10 - Aperçu de la liste des budgets .....	29
GUI 11 - Détail d'un budget .....	30
GUI 12 - Formulaire de création d'un budget .....	31

## Table des illustrations

---

GUI 13 - Tableau de bord .....	32
GUI 14 - Liste des dettes .....	33
GUI 15 - Formulaire de la dette .....	34

## 14 Annexes

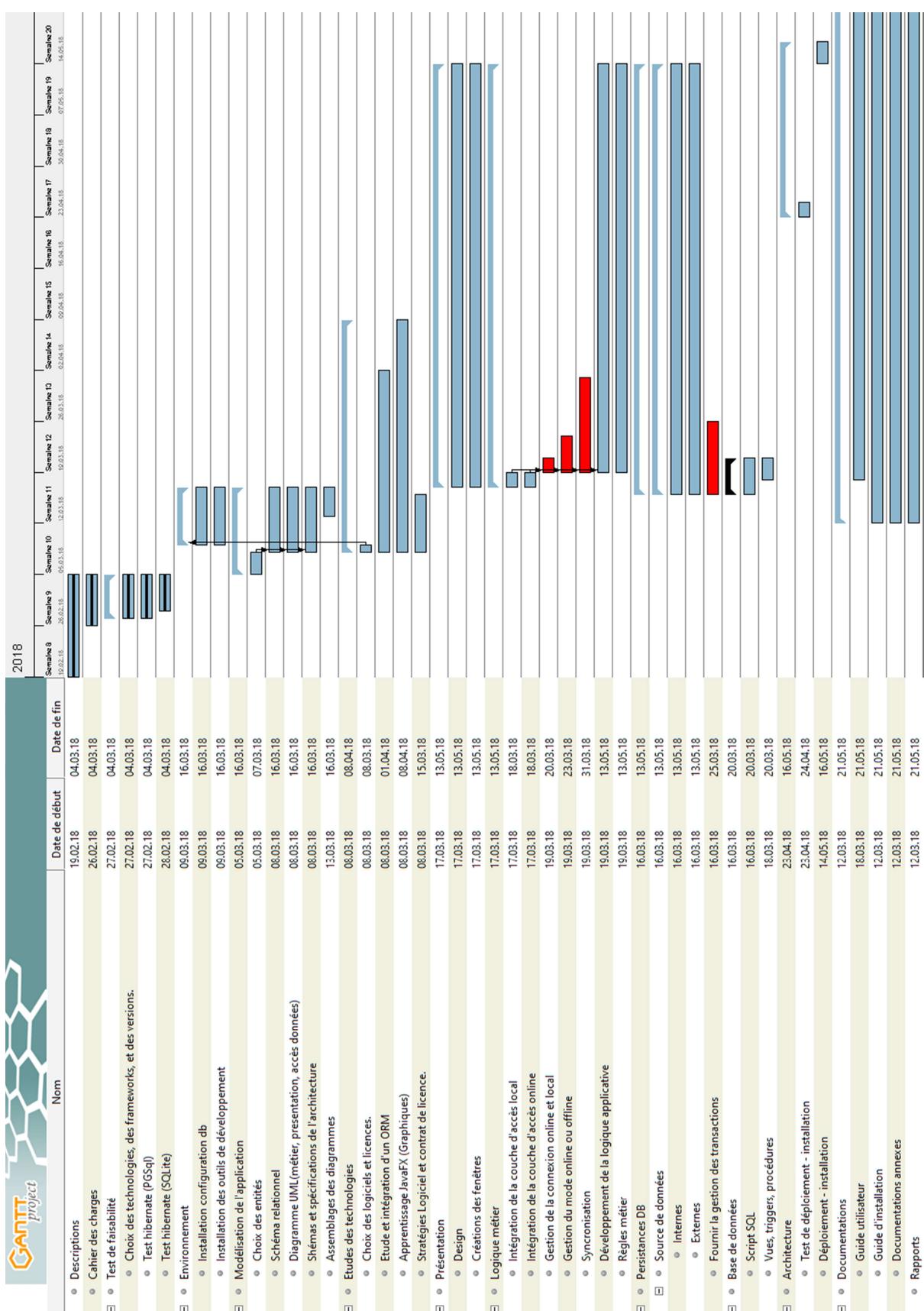
### 14.1 Cahier des charges initial du projet

Le cahier des charges initial est directement joint à la suite de ce rapport.

### 14.2 Planification du projet

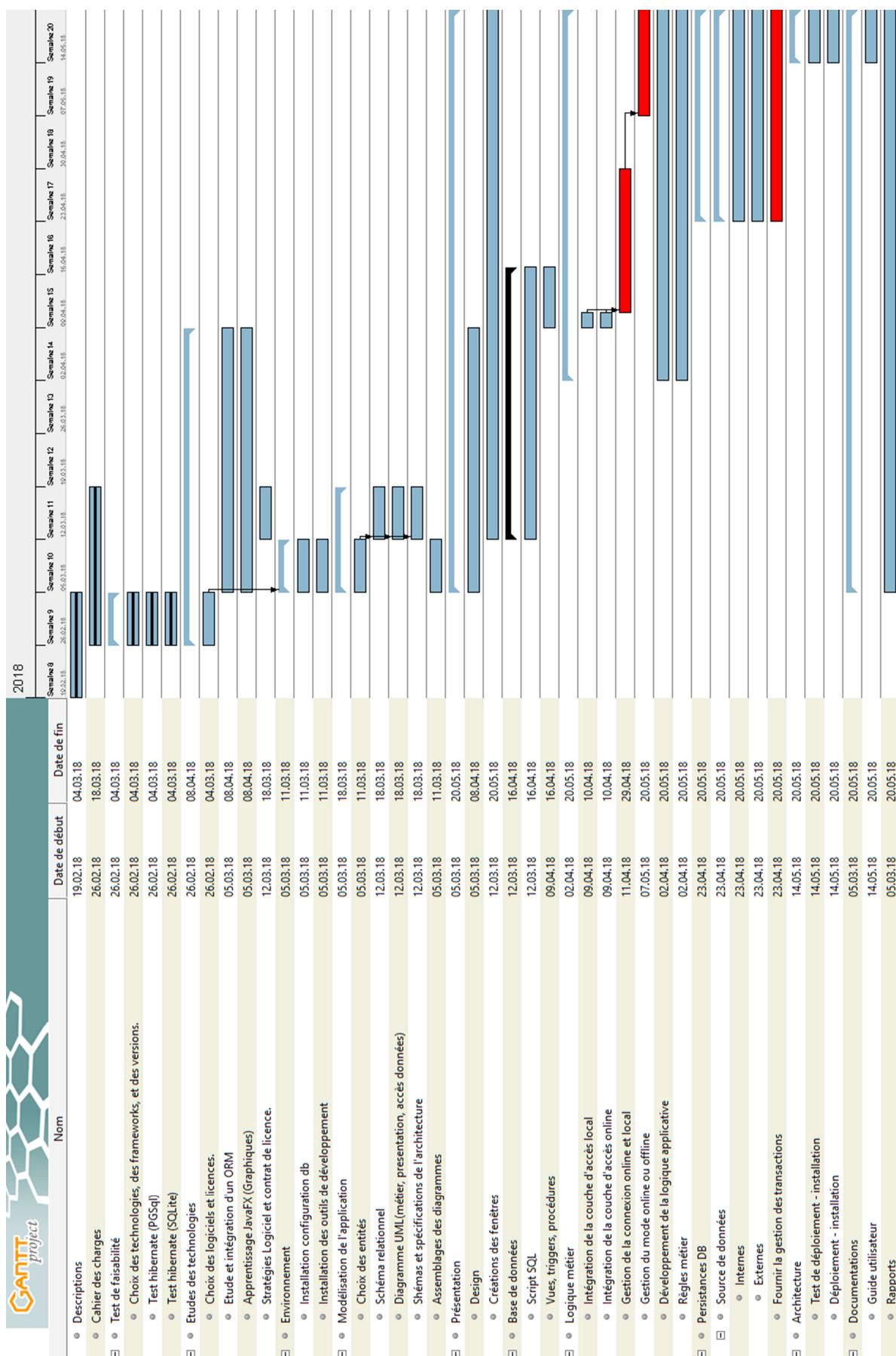
#### 14.2.1 Planification initiale

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	Total [h]
Bryan Curchod	1.5	12	7.5	7	7	7	6	6	6	6	6	6	6	6	6	97	
Daniel Gonzalez Lopez	5	14.5	9.5	8	7	7	6	6	6	6	6	6	6	6	7	106	
François Burgener	0.75	13.5	6	7	7	7	7	6	6	6	6	6	6	6	6	96.25	
Guillaume Zaretti	0.75	13.5	7	7	7	7	7	7	7	7	7	7	7	7	7	105.25	
Hélène Reymond	1.5	9.25	7	7.5	8	8	8	7	7	7	7	7	7	7	8	106.25	
Total [h]	9.5	62.8	37	36.5	36	36	35	32	32	32	32	32	32	32	34	0	510.75



## 14.2.2 Planification finale

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	Total [h]
<b>Bryan Curchod</b>	1	12	7.75	3.5	7.45	3.75	2.5	11	1.25	6.5	12.75	22	23				<b>114.45</b>
<b>Daniel Gonzalez Lopez</b>	4	16.5	4.75	2.5	7.25	3.25	1	10.5	1.5	4	12.75	15.5	24				<b>107.5</b>
<b>François Burgener</b>	0.75	13	6	8	6.5	3.75	2	4.5	1.25	8	18	15	21.75				<b>108.5</b>
<b>Guillaume Zaretti</b>	12	0	0	0	12.5	5.2	4	15.5	6	5.2	7.25	12	14.5				<b>94.1</b>
<b>Hélène Reymond</b>	1.5	9.25	4.25	0.5	3.25	3	2	4	3.25	10.5	12	18.25	35.25				<b>107</b>
<b>Total [h]</b>	<b>19.3</b>	<b>50.8</b>	<b>22.8</b>	<b>14.5</b>	<b>36.9</b>	<b>19</b>	<b>11.5</b>	<b>45.5</b>	<b>13.3</b>	<b>34.2</b>	<b>62.75</b>	<b>82.75</b>	<b>118.5</b>				<b>531.55</b>



### 14.2.3 Analyse des planifications

Pour commencer, nous avions mal évalué le nombre de semaines que nous avions à disposition. Nous n'avions pas fait attention au fait que dans le fichier Excel à remplir, les deux dernières semaines étaient réservées aux présentations et non pas pour le projet en soit.

Ensuite, nous pensions pouvoir tenir un rythme de travail régulier, mais c'était sans compter l'augmentation de la charge de travail dans les autres branches vers la moitié du semestre.

Pour en venir maintenant aux problèmes effectifs, la première chose que nous pouvons remarquer dans les planifications Gantt, c'est que nous avons commencé les tâches critiques beaucoup trop tard par rapport à ce qui était prévu, ce qui pour cette fois n'a pas posé de problème mais aurait pu être extrêmement dangereux.

La deuxième chose que l'on peut voir sur la planification finale, c'est que l'étude et l'intégration de l'ORM a pris plus de temps que prévu, ce qui a décalé le début des tâches critiques dont nous avons parlé précédemment.

De plus, nous pouvons remarquer que nous avons terminé énormément de tâches la dernière semaine, ce qui ne coïncide pas avec notre planification de base et qui aurait pu être critique si une erreur survenait au dernier moment. Nous aurions clairement dû faire en sorte d'avoir un minimum de tâches en cours la dernière semaine.

La dernière chose à remarquer est que notre planification surestimaient beaucoup de tâches. En effet, au début du projet nous n'avions pas les connaissances nécessaires pour bien les estimer. Nous pouvons donc remarquer que dans la planification de base, nous avons démarré beaucoup de tâches en parallèle sans prendre compte de leur charge de travail alors que dans la planification finale, et donc ce qui s'est réellement passé, nous avons des temps plus courts mais moins de tâches en parallèle.

### 14.3 Journaux de travail

Les journaux de travail de chacun sont joints à la suite de ce rapport au format Excel, pour une meilleure lisibilité.