

**A l'intention de M. René Rentsch**

Daniel Gonzalez Lopez, Guillaume Zaretti,

François Burgener, Bryan Curchod, Héléna Reymond

MoneyThoring

RAPPORT FINAL

FEVRIER 2018

Table des matières

[1 Introduction 4](#_Toc514418039)

[2 Objectifs du projet 5](#_Toc514418040)

[3 Concepts généraux 6](#_Toc514418041)

[4 Fonctionnalités implémentées 7](#_Toc514418042)

[4.1 Compte utilisateur 7](#_Toc514418043)

[4.2 Connexion sécurisée 7](#_Toc514418044)

[4.3 Compte bancaire 7](#_Toc514418045)

[4.4 Catégorie 7](#_Toc514418046)

[4.5 Transaction 7](#_Toc514418047)

[4.6 Dette 7](#_Toc514418048)

[4.7 Vue globale 8](#_Toc514418049)

[4.8 Budget 8](#_Toc514418050)

[4.9 Budgets partagés 8](#_Toc514418051)

[5 Conception et architecture du projet 9](#_Toc514418052)

[5.1 Technologies utilisées 9](#_Toc514418053)

[5.2 Architecture de la solution 9](#_Toc514418054)

[6 Bases de données (DB) 11](#_Toc514418055)

[6.1 Schéma relationnel 11](#_Toc514418056)

[6.2 PostgreSQL 12](#_Toc514418057)

[6.3 Derby 13](#_Toc514418058)

[6.4 Difficultés rencontrées 13](#_Toc514418059)

[7 Couche d’accès aux données (DAL) 14](#_Toc514418060)

[7.1 Projet template 14](#_Toc514418061)

[7.2 Mise en place de la DAL 19](#_Toc514418062)

[7.3 Hibernate 21](#_Toc514418063)

[7.4 Difficultés rencontrées 21](#_Toc514418064)

[8 Interface graphique (GUI) 22](#_Toc514418065)

[8.1 Menu latéral de l’application 22](#_Toc514418066)

[8.2 Fenêtre principale 23](#_Toc514418067)

[8.3 Fenêtre de connexion/enregistrement d’un compte utilisateur 24](#_Toc514418068)

[8.4 Fenêtre de compte bancaire 25](#_Toc514418069)

[8.5 Fenêtres des catégories 28](#_Toc514418070)

[8.6 Fenêtres des transactions 30](#_Toc514418071)

[8.7 Fenêtres des budgets 32](#_Toc514418072)

[8.8 Difficultés rencontrées 36](#_Toc514418073)

[9 Logique métier (BLL) 38](#_Toc514418074)

[9.1 Introduction 38](#_Toc514418075)

[9.2 Mappeurs 38](#_Toc514418076)

[9.3 Classe Authentication 39](#_Toc514418077)

[9.4 Difficultés rencontrées 39](#_Toc514418078)

[10 Tests 40](#_Toc514418079)

[10.1 Liste des tests effectués 40](#_Toc514418080)

[10.2 Bugs restants 40](#_Toc514418081)

[11 Conclusion 41](#_Toc514418082)

[11.1 Déroulement du projet 41](#_Toc514418083)

[11.2 Fonctionnement du groupe 41](#_Toc514418084)

[11.3 Avis personnel des membres du groupe 42](#_Toc514418085)

[11.4 Améliorations possibles 44](#_Toc514418086)

[12 Bibliographie / webographie 45](#_Toc514418087)

[12.1 BLL 45](#_Toc514418088)

[12.2 DAL 45](#_Toc514418089)

[12.3 DB 45](#_Toc514418090)

[12.4 GUI 45](#_Toc514418091)

[13 Table des illustrations 46](#_Toc514418092)

[13.1 Architecture 46](#_Toc514418093)

[13.2 Bases de données (DB) 46](#_Toc514418094)

[13.3 Couche d’accès aux données (DAL) 46](#_Toc514418095)

[13.4 Interface graphique (GUI) 46](#_Toc514418096)

[13.5 Logique métier (BLL) 47](#_Toc514418097)

[14 Annexes 48](#_Toc514418098)

[14.1 Cahier des charges initial du projet 48](#_Toc514418099)

[14.2 Planification du projet 48](#_Toc514418100)

[14.3 Journaux de travail 48](#_Toc514418101)

# Introduction

Avec MoneyThoring, nous proposons à l’utilisateur une gestion de ses transactions, qu’elles soient de simples dépenses ou revenus. Nous lui apportons aussi la possibilité de catégoriser ses transactions, de 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.

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.

Ces fonctionnalités concernant plusieurs utilisateurs, elles ne peuvent donc pas être gérées localement. Exception faite de ces dernières, tout est réalisable en local, avec ou sans connexion internet.

MoneyThoring propose un design poussé pour que l’information soit la plus claire, précise et que l’utilisateur n’ait pas à se creuser la tête pour comprendre son utilisation.

# Objectifs du projet

# Concepts généraux

# Fonctionnalités implémentées

## 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.

## 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 informations. Cette demande de connexion est un simple formulaire dans lequel l’utilisateur doit entrer son nom d’utilisateur et son mot de passe. L’application ne nécessite pas de double authentification.

Le mot de passe est haché, avec un sel ajouté, dans la base de données. Il n’est pas sauvé en clair.

## Compte bancaire

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.

## Catégorie

Il est possible de créer d’autres catégories que celles proposées par défaut. Une catégorie est définie par un nom et une couleur. Toutes les catégories, même celles proposées par défaut, peuvent être modifiées. 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 transaction, sa catégorie soit remplacée par celle par défaut.

## Transaction

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 (CHF par défaut).

Les transactions peuvent être modifiées ou supprimées si besoin.

## Dette

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.

## Vue globale

L’utilisateur peut en tout temps suivre l’évolution de son budget, 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. Le diagramme circulaire affiche les dépenses par catégorie.

## 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 le budget qu’il lui reste. 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 type, un nom, un montant et des catégories (aucune, une ou plusieurs). L’utilisateur peut modifier le montant, la devise, 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 à 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.

## 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. Dans ce cas-là les dépenses effectuées par l’utilisateur restent enregistrées dans le budget. Mais seul le créateur peut le supprimer totalement.

# Conception et architecture du projet

## Technologies utilisées

### Langage de programmation et librairie GUI

L’application a été implémentée en Java 9 avec l’environnement IntelliJ. 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.

### 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, 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, 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.

## 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.

# Bases de données (DB)

## 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 mais 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 créée 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.

## PostgreSQL

### 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 site 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. Le serveur se trouve en Amérique et de ce fait les temps de latence sont considérables.

Nous avons également ouvert un ticket chez Infomaniak 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.

### 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 renommons pas ces relations, puisque nous n’allons pas travailler dessus.

## 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/>.

Suite à l’utilisation de SQLite avec Hibernate, nous avons rencontré plusieurs problèmes pendant la mise en place du projet template. 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.

## Difficultés rencontrées

# Couche d’accès aux données (DAL)

## Projet template

La création d’un projet Template 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 premier projets template 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, notre environnement de développement. Ceci s’explique par le fait que NetBeans mette directement tous les fichiers de mapping et de configuration aux bonnes places avec toutes les dépendances nécessaires à la bonne utilisation d’Hibernate.

Nous avons donc créé ce projet template pour Hibernate sur IntelliJ, afin d’avoir les bonnes dépendances et configurations. La première chose que nous avons essayé de réaliser sur ce projet template 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 template nous avons fait un nouveau projet maven appelé MoneyThoring. 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 (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.pgsql.cfg.xml » pour le fichier de configuration d’Hibernate pour PostgreSQL
* « hibernate.pgsql.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

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 ci-dessous, qui utilise une session.



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 pour lors de la première exécution de ce code :



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 ceci au pom.xml :



DAL 5 - Dépendances maven ajoutées au pom.xml

Nous avons dû spécifier le fichier xml de configuration dans le « session configure », 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.



DAL 6 - Configurations d'Hibernate manquantes

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



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 :



DAL 8 - Dépendance maven ajoutée au pom.xml



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 :



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 :



DAL 10 - pom.xml



DAL 11 - hibernate.pgsql.cfg.xml

## Mise en place de la DAL

La couche d’accès aux données encapsule les mécanismes propres à la persistance des données, elle 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

### 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.

### dal.ientities

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.

### 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 « ientities ». 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 connaitre 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.

#### dal.entities.derby

Ce package contient l’implémentation des interfaces dal.ientities pour le mapping objet Derby. Le modèle objet Derby est tagué avec les annotations d’Hibernate, afin de mapper le 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.

#### dal.entities.pgsql

Idem que pour Derby, ce package contient l’implémentations des interfaces dal.ientities 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.

### dal.irepositories

Ce package contient les interfaces qui permettent d’effectuer des opérations en base de données tel que : création, modification, suppression ainsi que d’autre 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 raison que ces interfaces n’utilisent aucune implémentation, mais uniquement les interfaces contenues dans dal.ientities.

#### dal.repository.derby

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

#### dal.repository.pgsql

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

### 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 même 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.

### 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 au donnée et les couches plus hautes.

## 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 de base ou des types plus évolués (leur équivalent en objet). Nous avons donc choisi de travailler avec des types de bases. 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.

## Difficultés rencontrées

# Interface graphique (GUI)

## 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 (Évolution possible : remplacer les boutons par une Select List ou équivalent).

## Fenêtre principale



GUI 1 - Fenêtre principale, menu fermé 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.

### Composition

Voici comment le fichier FXML (le modèle) 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. : "Dashboard", "Budget – Nourriture", etc.)
* CENTRE : la partie centrale du BorderPane contient un second AnchorPane (choix non définitif, il faut voir comment cet élément se comportera une fois les autres vues développées) qui permettra de charger le contenu voulu (exemple : la vue du Dashboard ou la liste des budgets). Le centre contient également un JFXDrawer qui est le conteneur de notre menu latéral.

### Contrôleur

Le contrôleur permet de définir le comportement des éléments d'un FXML. Pour la fenêtre principale il faut pouvoir gérer l'ouverture et la fermeture du menu latéral, ainsi que le chargement de la vue choisie.

D’une fois que nous avions plusieurs vues actives, il a fallu adapter l'implémentation du chargement des vues dans le contrôleur de la fenêtre principale. Pour pouvoir gérer plusieurs vues efficacement, nous avons défini un tableau de Titre (String) contenant le nom des vues, et un tableau de vue (String, ce sont les chemins vers les fichiers) pour récupérer facilement les fichiers. Cependant lors du chargement nous ne pouvons pas nous débarrasser d'un switch-case, car nous devons définir le contrôleur sur le moment du chargement.

## Fenêtre de connexion/enregistrement d’un compte utilisateur

### Composition



GUI 3 - Fenêtre de connexion à MoneyThoring

Pour définir notre interface graphique nous utilisons le format de fichier FXML. Le FXML est le format utilisé par JavaFX pour définir des interfaces graphiques. Il utilise tout simplement la syntaxe XML.

Pour cette vue, qui contient plusieurs éléments, nous utilisons le fichier « LoginRegister.fxml ». Le conteneur principal de celle-ci est un Pane, qui est la classe de base pour les panneaux de mise en page. Il permet d’y insérer des éléments graphiques (conteneur, contrôle, composant, etc.) et de les positionner librement. Tout élément ajouté à ce conteneur est appelé « enfant » de ce dernier. Nous en avons trois : un label qui dispose le titre de notre vue, un GridPane pour l’espace de connexion et un GridPane pour l’espace d’enregistrement.

On utilise des GridPane pour les espace de connexion et d’enregistrement, car ils permettent d’ajouter des enfants dans une grille flexible composées de lignes et de colonnes. 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 ajoutés. Les composants utilisés dans nos GridPane pour la récupération d’une entrée utilisateur (email, nom, prénom, mot de passe) sont des Textfield et Passwordfield. Et pour ce qui est des contrôles nous utilisons des boutons.

### Contrôleur

Dans notre fichier « Controller\_loginRegister.java » nous avons deux événements qui portent sur les deux boutons de l’espace de connexion et d’enregistrement. Chacun de ces événements appellent une méthode de la BLL (Business Logical Layer).

Un événement « clickLoginButton » appelle la méthode « login », qui vérifie si l’email est bien dans la base de données et que le mot de passe correspond bien à l’email spécifié. Si ce n’est pas le cas, la méthode retourne une erreur. Nous traitons cette erreur par l’affichage d’un message d’erreur à l’utilisateur : « Identifiant ou mot de passe incorrect ».

Un autre événement « clickRegisterButton » appelle la méthode « register », qui vérifie si l’email est déjà dans la base de données. Si l’email est déjà présente dans la base de données, cette méthode retourne une erreur. Si le mot de passe et sa confirmation ne sont pas identiques nous générons également une erreur. Nous traitons ces erreurs par le soulignement en rouge des champs mal renseignés.

## Fenêtre de compte bancaire

### Vue principale

#### Composition



GUI 4 - Fenêtre des comptes bancaires

Pour cette vue nous utilisons le fichier « bankAccount.fxml ». Cette vue possède comme conteneur principal un AnchorPane. Ce conteneur nous permet de fixer le bord de ses enfants à une certaine distance son propre bord. Cela est pratique pour le redimensionnement de notre fenêtre.

Dans ce conteneur nous avons deux enfants :

* Un FlowPane pour lister les comptes bancaires
* Un bouton pour la création d’un compte bancaire

Nous utilisons un FlowPane pour lister les comptes bancaires, car il nous permet à l’ajout d’un enfant, de ne pas devoir spécifier la position ce dernier. Il se place automatiquement en haut à gauche (par défaut). Ainsi l’ajout de plusieurs enfants entraîne un positionnement des enfants sur une seule ligne. Et une fois la ligne de notre conteneur remplie, les éléments suivants sont décalés à la suite sur la ligne suivante. Lorsque nous redimensionnons notre fenêtre, les enfants qui sont dans le FlowPane se repositionnent de telle sorte qu’il y en a un maximum sur une ligne (voir image). Pour ce qui est du contrôle pour l’ajout d’un compte bancaire, nous utilisons un bouton.

#### Contrôleur

Notre classe « Controller\_bankAccount » implémente l’interface « Initializable », qui nous permet d’avoir la méthode « initialise ». Cette méthode sert à initialiser notre contrôleur. Dans notre cas nous récupérons la liste des comptes via la méthode « getBankAccounts » de la classe « ClientLogic ». Pour chacun de ces comptes, nous créons un AccountDisplayer que nous ajoutons à notre FlowPane via la méthode « addToFrame ».

De plus nous ajoutons une action à notre bouton, « createBankAccount\_button » qui appelle la méthode « createButton ». Cette dernière charge la vue de création d’un compte bancaire. Lorsque l’utilisateur a rempli tous les champs et validé sa création, le contrôleur « Controller\_createBankAccount » créée l’instance de la classe « BankAccountLogic » et appelle « add » avec comme paramètre l’instance de notre « BankAccountLogic ». Cette méthode « add » fait le même procédé que pour l’initialisation. Nous créons un « AccountDisplayer » avec l’instance de notre classe « BankAccountLogic » que nous ajoutons à notre conteneur.

Un « AccountDisplayer » est un GridPane qui contient le nom attribué à notre compte bancaire, le montant de notre compte bancaire, ainsi que la date de notre dernière transaction. De plus cette classe possède une action qui permet, lorsque nous cliquons dessus, de charger la fenêtre sur le détail de notre compte bancaire via la méthode « detailBankAccount ».

### Vue détaillée

#### Composition



GUI 5 - Vue détaillée d'un compte bancaire

Pour cette vue nous utilisons le fichier « detailBankAccount.fxml ». Elle possède comme conteneur principal un AnchorPane.

Dans ce conteneur nous avons deux enfants :

* Un AnchorPane pour afficher les informations d’un compte bancaire
* Un lineChart pour voir l’évolution du solde dans le mois en cours

Nous utilisons un lineChart (graphique en courbe), afin d’avoir une vision sur l’évolution du solde d’un compte bancaire.

#### Contrôleur

Notre classe « Controller\_detailBankAccount » implémente l’interface « Initializable », qui nous permet d’avoir la méthode « initialize ». Nous utilisons cette méthode pour ajouter les informations aux différent champs (nom de la banque, type du compte, date de la dernière transaction, le solde du compte et les données du graphiques), lors de la création d’une instance de la classe « Controller\_detailBankAccount ».

Nous avons aussi un évènement sur le bouton « returnButton » qui aura comme effet de retourner sur la page principale des comptes bancaires. Cet évènement appellera la méthode XXX.

### Formulaire des comptes bancaires

#### Composition

#### Contrôleur

## Fenêtres des catégories

### Vue globale



GUI 6 - Aperçu de la liste des catégories dans Scene Builder

La vue de la liste des catégories permet à l'utilisateur de consulter ses catégories, et d'en créer s'il en ressent le besoin.

#### 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. Cet objet gère automatiquement l'espacement entre chaque élément, ainsi que leur emplacement dans la fenêtre (retour à la ligne). Nous avons également ajouté un AnchorPane pour accueillir le contenu du formulaire.

#### Contrôleur

Dans le contrôleur de la liste des catégories (« Controlleur\_categoryList ») nous avons ajouté une classe interne « CategoryDisplayer » qui s'occupe de l'affichage et de la mise en forme d'une catégorie.

### Formulaire de création d’une catégorie



GUI 7 - Aperçu du formulaire dans Scene Builder

#### Composition

L'élément racine est un AnchorPane, afin de pouvoir fixer les dimensions du contenu. Puis nous avons une VBox (pour centrer le contenu verticalement) qui lui, contient un second AnchorPane. Dans ce dernier nous avons nos boutons et une HBox (pour centrer le contenu horizontalement) dans lequel nous trouvons les champs du formulaire, à savoir un TextField et un ColorPicker.

#### Contrôleur

## Fenêtres des transactions

### 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.

#### 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, tous deux contiennent au centre une JFXTreeView qui permet de lister nos transactions.

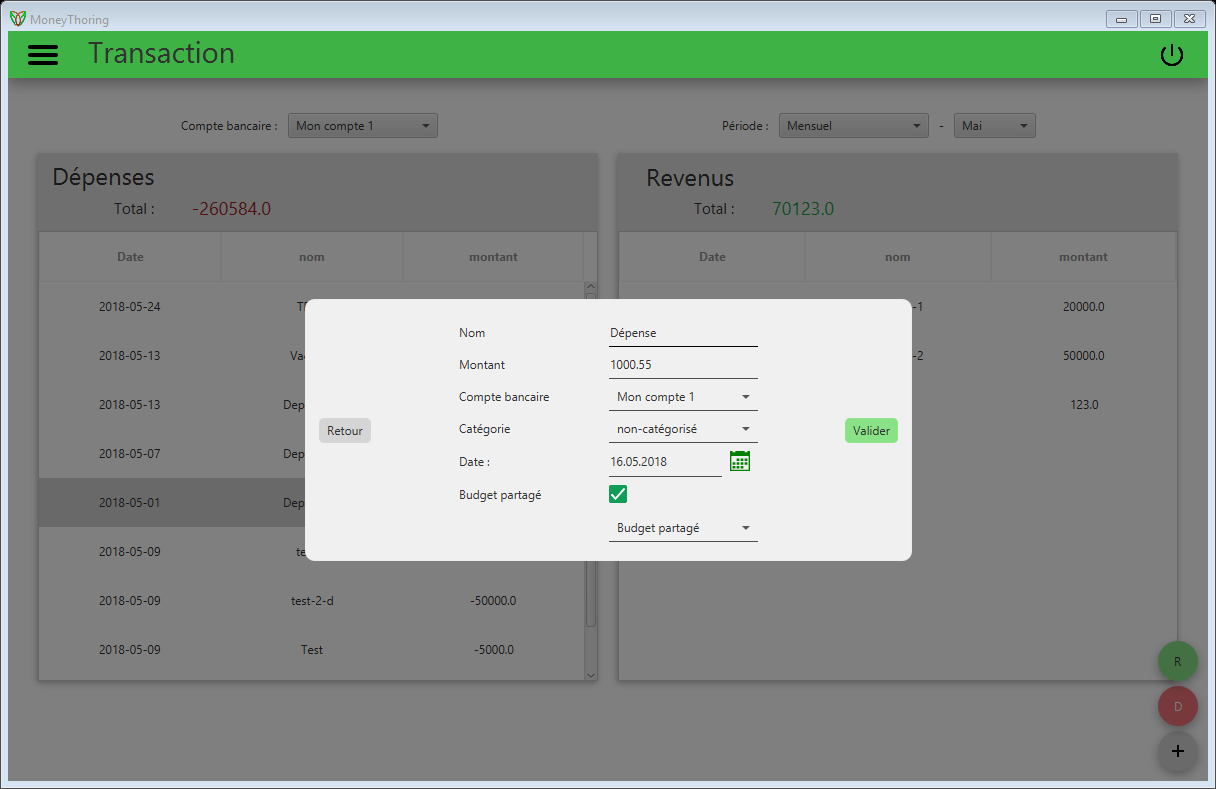
Pour l’ajout d’une dépense ou d’un revenu nous utilisons une JFXNodeListe qui permet d’avoir une liste de boutons (une pour les revenus et une pour les dépenses). Cette JFXNodeListe nous permet lorsque l’on clique sur un bouton d’en afficher deux.

#### Contrôleur

Dans le contrôleur qui liste des transactions (« Controlleur\_listTransaction ») nous utilisons l’interface « Initializable » pour initialiser nos différents contenus, évènements de notre fenêtre. Dans ce contrôleur nous gérons tous les événements liés au 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, qui eux vont appeler la méthode « callForm » qui va charger le formulaire des transactions.

Pour la modification/suppression d’une transaction nous utilisons un événements double clic sur la transaction, afin que le formulaire des transactions se charge. Sur ce formulaire nous avons le choix de modifier les champs et ensuite de valider la modification, ou alors de supprimer la transaction via le bouton supprimer.

### Formulaire d’ajout/modification/suppression des transactions



GUI 9 - Formulaire des transactions

#### Composition

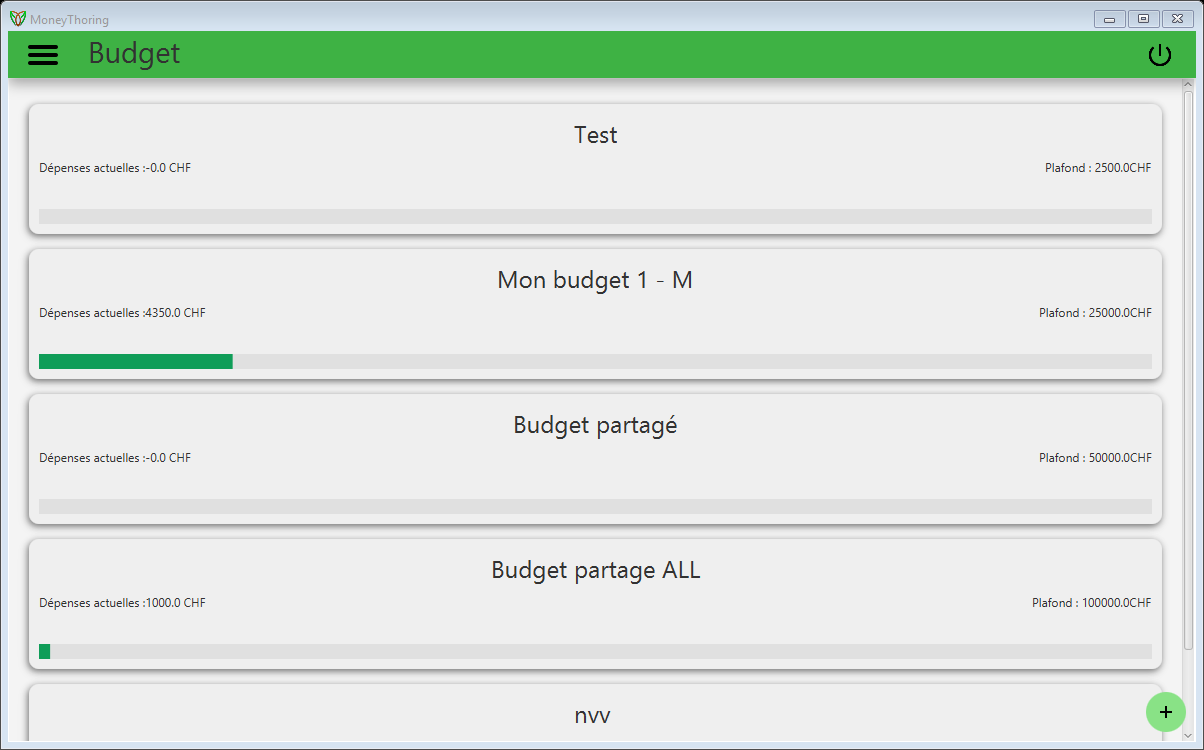
L'élément racine est un AnchorPane, afin de pouvoir fixer les dimensions du contenu. Puis nous avons un VBox (pour centrer le contenu verticalement) qui lui, contient un second AnchorPane. Dans ce dernier nous avons nos boutons et un GridPane dans lequel nous trouvons les champs du formulaire, à savoir des Textfield 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.

#### Contrôleur

Dans le contrôleur « Controlleur\_formTransaction » nous récupérons tous les champs insérés par l’utilisateur. Nous vérifions que les entrées sont correctes, et nous créons ou modifiant une transaction via la méthode « formValidation » appelé dans l’événements du bouton « Valider ». Nous avons aussi un bouton « Supprimer » qui elle appelle une méthode de la BLL qui supprime la transaction de la base de données.

## Fenêtres des budgets

### Vue globale



GUI 10 - Aperçu de la liste des budgets

Cette vue permet de lister les budgets, une autre grande fonctionnalité de notre logiciel.

#### Composition

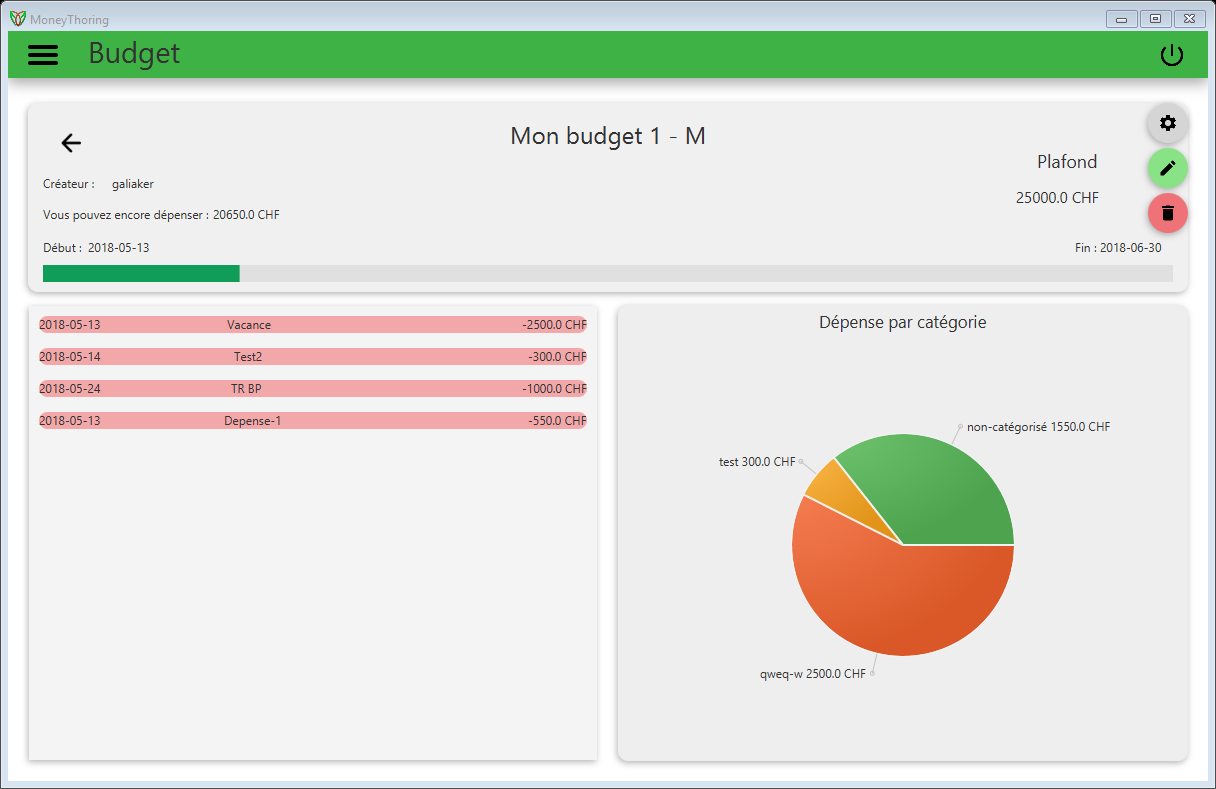
L'élément racine est un AnchorPane, afin de pouvoir redimensionner le contenu convenablement. A l'intérieur nous trouvons un ScrollPane qui possède une VBox pour faire en sorte que les budgets s'ajoutent verticalement (comme une liste). Pour l’affichage du budget nous utilisons un fichier fxml « budgetDIsplayer » qui contient comme élément racine un AnchorPane, qui lui possède des labels pour le nom du budget ainsi que le plafond et la dépense actuelle. Pour l’évolution des dépenses du budget nous utilisons une ProgressBar.

#### Contrôleur

Dans le contrôleur « Controlleur\_listBudget » nous utilisons les mêmes procédés que dans les autres contrôleurs qui listent des éléments. Nous utilisons l’interface « Initializable », afin d’initialiser tous nos éléments ainsi que nos évènements. L’élément sur un des boutons de notre JFXNodeList appellera la méthode « callForm » qui lui, chargera notre formulaire des budgets.

Dans ce contrôleur nous implémentons directement (en classe interne) le contrôleur de notre « budgetDIsplayer.fxml ». Ce displayer a un événement pour lequel que lorsque l’on clique dessus, la méthode « openDetail » est appelée pour charger la fenêtre d’un détail d’un budget.

### Fenêtre du détail d’un budget



GUI 11 - Détail d'un budget

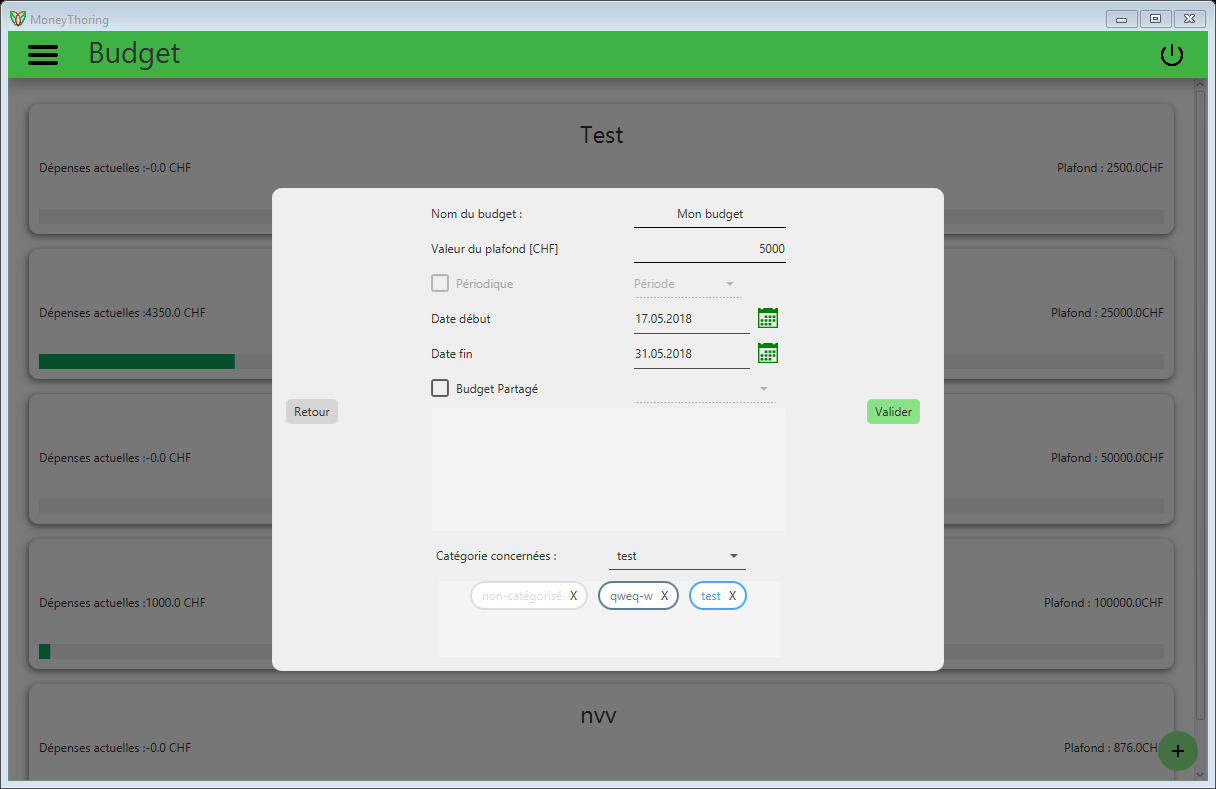
#### Composition

L'élément racine est un AnchorPane, afin de pouvoir redimensionner le contenu convenablement. A l'intérieur nous trouvons un AnchorPane pour les informations générales du budget comme le titre du budget, le créateur, combien il peut encore dépenser, le plafond, la date de début et de fin ainsi que les boutons d’édition et de création. Nous avons aussi un GridPane qui lui, contient un ScrollPane pour la liste des transactions et un AnchorPane qui lui contient notre graphique en camembert.

#### Contrôleur

Dans le contrôleur « Controlleur\_detailBudget » nous utilisons aussi l’interface « Initializable » qui nous permet ici d’initialiser les informations générales de notre budget (nom, créateur, dépense, plafond ainsi que la date de début/fin). Il nous permet aussi de lister toutes les dépenses liées aux catégories de ce budget dans l’intervalle des date (début/fin). Si le budget n’est lié à aucune catégorie nous prenons toutes les dépenses liées à tous les comptes bancaires pour l’intervalle de date(début/fin) tous les dépenses des catégories liées à ce budget. Pour l’édition d’un budget, nous utilisons la méthode « modifyItem » qui elle charge le formulaire avec les informations. Lors de la validation du formulaire nous rafraichissons toutes les informations de la page de détail d’un budget.

### Formulaire d’un budget



GUI 12 - Formulaire de création d'un budget

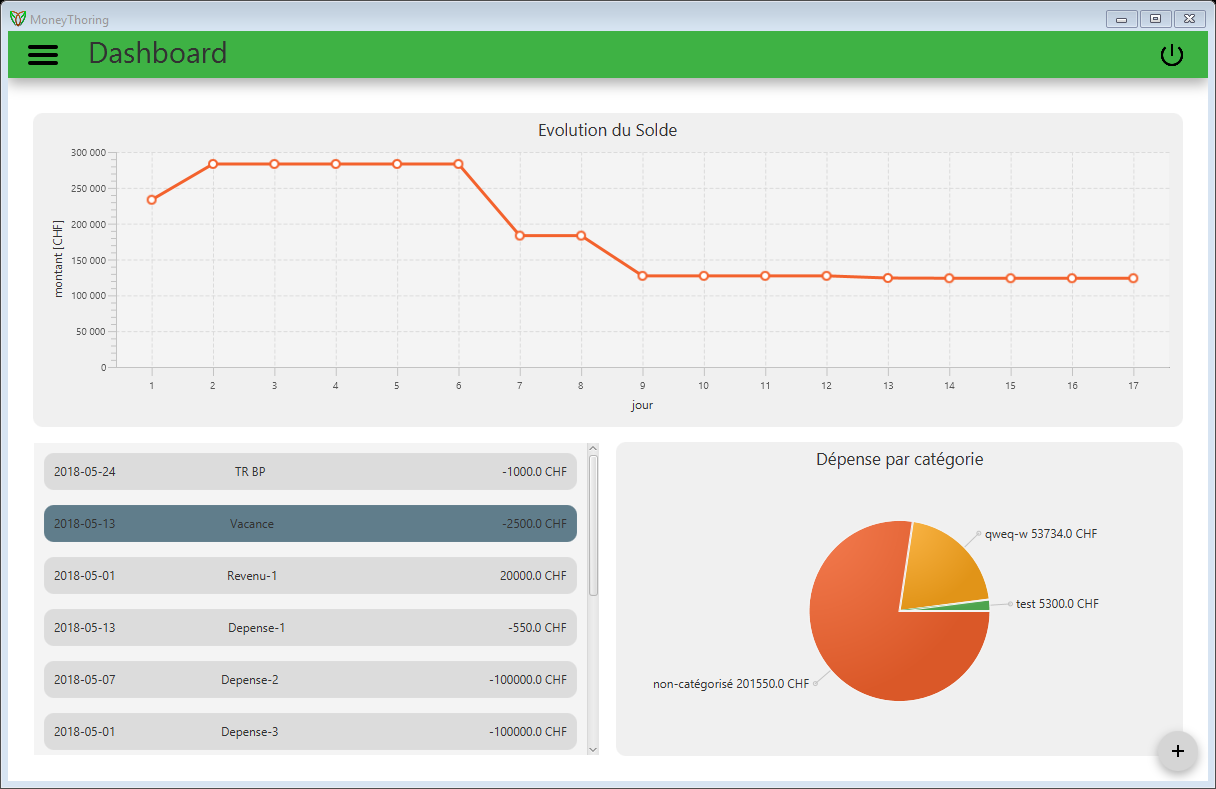
#### Composition

L'élément racine est un AnchorPane, afin de pouvoir fixer les dimensions du contenu. Puis nous avons une VBox (pour centrer le contenu verticalement) qui lui, contient un second AnchorPane. Dans ce dernier nous avons nos boutons et un GridPane dans lequel nous trouvons les champs du formulaire, à savoir 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 ce dernier est sélectionné, nous pouvons choisir des utilisateurs liés à ce budget via une ComboBox. Les utilisateurs s’affichent dans l’AnchorPane juste en dessous. Nous pouvons faire de même pour les catégories. Nous pouvons sélectionner des catégories via un ComboBox, qui s’affichent juste en dessous dans un AnchorPane.

#### Contrôleur

Dans le contrôleur « Controlleur\_formBudget » nous récupérons tous les champs insérés par l’utilisateur, nous vérifions que les entrées soient correctes, et nous créons ou modifiant un budget via la méthode « formValidation » appelée dans l’événements du bouton « Valider ».

### Fenêtre du tableau de bord



GUI 13 - Tableau de bord

#### Composition

L'élément racine est un AnchorPane, afin de pouvoir redimensionner le contenu convenablement. A l'intérieur nous trouvons un GridPane qui lui contient trois AnchorPane afin de contenir, un LineChart, un ScrollPane et un PieChart. A l’intérieur du ScrollPane nous avons une VBox afin de lister nos transactions Nous avons aussi un bouton qui nous permet de créer une transaction depuis le tableau de bord.

#### Contrôleur

Dans le contrôleur « Controlleur\_dashboard » nous utilisons l’interface « Initializable » afin d’initialiser nos différents éléments. Pour l’insertion des valeurs dans le graphique en courbe, nous utilisons la méthode « setDataLineChart » qui elle récupère toutes les transactions liées au compte par défaut du mois courant et les affiche. Pour lister les transactions nous utilisons un « transactionDisplayer ». Lors de la création de cet objet nous l’ajoutant à notre VBox. Pour insérer les valeurs dans notre graphique en camembert nous utilisons la méthode « setDataPieChart » qui elle calcule les dépenses liées à chacune des catégories et les insérer dans notre graphique.

## Difficultés rencontrées

### Utilisation de la librairie JFoenix sous IntelliJ IDEA

Le SceneBuilder proposé par IntelliJ 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/>

### 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).

### 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.

### 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 « getRessource » 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.

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

Nous arrivions à appeler la fenêtrede formulaire, mais nous ne pouvions pas récupérer l'information crée 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).

### 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.

# Logique métier (BLL)

## Introduction

* Code BLL séparé en 3 packages : logic, dal.mappers, model

## Mappeurs

### 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.

### 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 Client.

Avant chaque nouvelle insertion dans la base de données, un objet de type ClientModel est créé au niveau de la BLL. Celui-ci représente l’entité Client au format Java. Pour envoyer ce client en base de données, la BLL doit appeler une méthode spécifique de la DAL qui attend un ClientModel en paramètre et qui retourne un IDALClientEntity. Cette dernière entité représente le client au format base de données. Une fois cette entité reçue, la BLL doit l’envoyer à la méthode « addClient » d’un objet de type IClientRepository, 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 un IDALClientEntity. Celui-ci représente l’entité Client au format base de données. Pour intégrer ce client, la BLL doit appeler une autre méthode de la DAL qui attend un IDALClientEntity en paramètre et qui retourne un ClientLogic (héritant de ClientModel). Ce dernier représentant l’entité Client au format Java, la BLL peut donc interagir avec et l’intégrer à l’application.

## Classe Authentication

## Difficultés rencontrées

# Tests

## Liste des tests effectués

## Bugs restants

# Conclusion

## Déroulement du projet

## Fonctionnement du groupe

## Avis personnel des membres du groupe

### Bryan Curchod

### Daniel Gonzalez Lopez

### 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.

### 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.

### Héléna 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 à gérer. 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’utiliserai dans le futur si l’occasion se représente.

Pour terminer je tiens à signaler que nous avons rencontrés beaucoup de 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.

## Améliorations possibles

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 couche, 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. Coté 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.

# Bibliographie / webographie

## BLL

### Gestion des SQL dates

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

### 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>

## DAL

### 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>

## DB

### Création des scripts et requêtes

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

## GUI

### JavaFx

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

### Jfoenix

* <http://www.jfoenix.com>
* <https://www.youtube.com/watch?v=22QlOj6JVe4&list=PLhs1urmduZ29LNYi_MaoU60JemQ6Aei6A>

### Icônes

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

# Table des illustrations

## Architecture

[Architecture 1 - Schéma général de l'architecture de MoneyThoring 8](#_Toc514403792)

## Bases de données (DB)

[DB 1 - Schéma relationnel de la base de données 10](#_Toc514403776)

## Couche d’accès aux données (DAL)

[DAL 1 - Importation du schéma de base de données PostgreSQL 14](#_Toc514403764)

[DAL 2 - Arborescence de projet après la configuration DB de PostgreSQL 14](#_Toc514403765)

[DAL 3 - Code de test de l'intégration d'Hibernate 15](#_Toc514403766)

[DAL 4 - Résultat de l'exécution du premier test d'Hibernate 15](#_Toc514403767)

[DAL 5 - Dépendances maven ajoutées au pom.xml 16](#_Toc514403768)

[DAL 6 - Configurations d'Hibernate manquantes 16](#_Toc514403769)

[DAL 7 - Résultat de l'exécution du troisième test d'Hibernate 16](#_Toc514403770)

[DAL 8 - Dépendance maven ajoutée au pom.xml 17](#_Toc514403771)

[DAL 9 - Mise à jour du driver utilisé pour PostgreSQL 17](#_Toc514403772)

[DAL 10 - pom.xml 17](#_Toc514403773)

[DAL 11 - hibernate.pgsql.cfg.xml 18](#_Toc514403774)

[DAL 12 - Structure de la couche d'accès aux données 18](#_Toc514403775)

## Interface graphique (GUI)

[GUI 1 - Fenêtre principale, menu fermé GUI 2 - Fenêtre principale, menu ouvert 22](#_Toc514416157)

[GUI 3 - Fenêtre de connexion à MoneyThoring 23](#_Toc514416158)

[GUI 4 - Fenêtre des comptes bancaires 24](#_Toc514416159)

[GUI 5 - Vue détaillée d'un compte bancaire 25](#_Toc514416160)

[GUI 6 - Aperçu de la liste des catégories dans Scene Builder 27](#_Toc514416161)

[GUI 7 - Aperçu du formulaire dans Scene Builder 28](#_Toc514416162)

[GUI 8 - Aperçu de la liste des transactions 29](#_Toc514416163)

[GUI 9 - Formulaire des transactions 30](#_Toc514416164)

[GUI 10 - Aperçu de la liste des budgets 31](#_Toc514416165)

[GUI 11 - Détail d'un budget 32](#_Toc514416166)

[GUI 12 - Formulaire de création d'un budget 33](#_Toc514416167)

[GUI 13 - Tableau de bord 34](#_Toc514416168)

## Logique métier (BLL)

**Aucune entrée de table d'illustration n'a été trouvée.**

# Annexes

## Cahier des charges initial du projet

## Planification du projet

### Planification initiale



* Pas sur 15 semaines mais sur 13, donc heures erronées ??
* Mal évalué le temps à cause des autres cours

### Planification finale

### Analyse des planifications

## Journaux de travail

### Bryan Curchod

### Daniel Gonzalez Lopez

### François Burgener

### Héléna Line Reymond