Moneythoring

Rapport

HEIG-VD

Projet de semestre

Table des matières

1 Descriptif 2

2 Fonctionnalités (A METTRE A JOUR A LA FIN DU RAPPORT) 3

2.1 Fonctionnalités de base 3

2.2 Fonctionnalités optionnelles 5

3 Architecture (A METTRE A JOUR) 7

4 Choix d’implémentation 8

4.1 Langage de programmation et GUI 8

4.2 Framework 8

5 Bases de données (DB) 9

5.1 Schéma relationnel 9

5.2 PostgreSQL 10

5.3 Derby 11

6 Couche d’accès aux données(DAL) 12

6.1 Projet template 12

6.2 Mise en place de la DAL 17

6.3 Hibernate 18

7 Interface graphique (GUI) 19

7.1 Menu latéral de l’application 19

7.2 Fenêtre principale 19

7.3 Fenêtre de connexion/enregistrement d’un compte utilisateur 20

7.4 Fenêtre de compte bancaire 21

7.5 Fenêtre des listes de catégories 24

7.6 Fenêtre de la liste des transactions 25

7.7 Problèmes rencontrés 25

8 Logique métier (BLL) 27

8.1 Introduction 27

8.2 Classe Authentication 27

9 Conclusion 28

# Descriptif

Avec MoneyThoring, nous proposons à l’utilisateur une gestion de ses transactions, qu’elles soient de simples dépenses ou revenus, des virements entre comptes, des dettes ou encore des factures. Nous lui apportons aussi la possibilité de catégoriser ses transactions, de créer des objectifs de budget pour pouvoir suivre les 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 (fonctionnalité en ligne) ou de travailler uniquement en local (hors ligne). Selon ce choix, certaines fonctionnalités sont accessibles ou pas. Bien sûr, même si l’utilisateur possède un compte, s’il vient à manquer d’une connexion internet, il peut sans autre continuer son travail hors ligne. Ce dernier étant synchronisé une fois la connexion avec la base de données en ligne rétablie.

Certaines fonctionnalités ne sont disponibles qu’avec la création d’un compte utilisateur et d’une connexion à internet. Ces fonctionnalités concernent effectivement plusieurs utilisateurs et ne peuvent donc pas être gérées localement. Exception faite de ces dernières, tout est réalisable en local, avec ou sans compte utilisateur.

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.

# Fonctionnalités (A METTRE A JOUR A LA FIN DU RAPPORT)

## Fonctionnalités de base

### Création compte utilisateur

Si l’utilisateur ne possède pas de compte à l’ouverture de l’application, il peut choisir d’en créer un ou de continuer sans. En choisissant la première option, qui n’est accessible que si l’utilisateur à une connexion internet, 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, un email de validation contenant un code est envoyé. Une fois le code saisi dans l’application, le compte est activé.

En choisissant la deuxième option, l’utilisateur peut sans autre utiliser l’application mais n’aura pas accès aux fonctionnalités partagées entre utilisateurs.

### Connexion sécurisée

Pour les utilisateurs possédant un compte validé, une connexion est exigée au démarrage de l’application. Cette connexion évite qu’un tiers puisse modifier les informations, même en travaillant en hors ligne (les données étant synchronisées par la suite). 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 (optionnel), le montant actuel, un pourcentage d’intérêt éventuel et s’il faut l’utiliser comme compte par défaut lors de transactions. Toutes ces informations, exceptée le solde du compte qui est modifié automatiquement par les transactions, peuvent être modifiées par la suite.

Les comptes bancaires peuvent également être supprimés. L’utilisateur peut alors choisir de virer le solde vers un autre compte ou de perdre ce dernier.

### 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 et supprimées.

### Devise

Une conversion automatique est faite lorsque les transactions effectuées sont dans une autre devise que CHF, en fonction du taux du jour. Cette fonctionnalité nécessite une connexion internet, le logiciel devant aller chercher les taux sur internet.

### 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 sélectionné par l’utilisateur lors de sa création), une devise utilisée, le taux de change de la journée et son type (revenu ou dépense).

Les transactions peuvent également avoir une notion de récurrence (salaire et factures). Celle-ci peut être annuelle, mensuelle ou plus spécifique (2 semaines, 2 mois, etc.) et s’exécuter à une date précise, par exemple tous les 25 du mois.

Les transactions peuvent être supprimées ou modifiées dans le cas d’erreurs.

### Virement compte à compte

Il est possible d’enregistrer des virements entre les différents comptes d’un utilisateur. Un virement est pris en compte comme une transaction. Il y a donc une dépense (transaction sortante) pour le compte à débiter et un revenu (transaction entrante) pour le compte à créditer.

### 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, un intérêt, une date limite et une description. Pour les dettes synchronisées, il est possible de spécifier un nom d’utilisateur.

Une fois une dette acquittée, l’utilisateur peut la valider et la transaction qui en découle est automatiquement ajoutée. Dans le cas de dettes synchronisées, il est nécessaire que les deux partis confirment la dette, à la réception et à l’acquittement.

### Vue globale

L’utilisateur peut en tout temps suivre l’évolution de son budget, grâce à une vue globale (depuis le début du budget jusqu’à maintenant), annuelle ou mensuelle. Cette vue permet de comparer de manière graphique les différentes dépenses et rentrées d’argent. En plus d’être filtrée chronologiquement, elle peut également l’être selon le mode d’affichage ; graphique en courbe, diagramme circulaire, diagramme en barres.

#### Graphique en courbe

Le graphique en courbe permettra de voir rapidement et simplement l’évolution des comptes.



#### Diagramme circulaire

Deux diagrammes circulaires affichent les dépenses et les entrées par catégorie. Les pourcentages exacts sont ajoutés à côté du graphique.

### Budget

Un budget se caractérise en deux types : les budgets récurrents (en fonction des catégories) et les budgets ponctuels (en fonction de deux dates données début/fin). Ces budgets permettent de voir l’évolution des dépenses dans un domaine précis, tout en permettant à l’utilisateur d’être notifié quand il s’approche de la limite fixée. Les budgets sont représentés de manière graphique sur une ligne, avec le budget qu’il lui reste (vert par exemple) et les dépenses déjà effectuées (rouge). 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). Lors de l’ajout du montant, l’utilisateur peut choisir une devise différente que celle par défaut. Si le budget est ponctuel, une durée est demandée (date de début et de fin).

Il est possible de supprimer n’importe quel budget. L’utilisateur peut modifier le montant, la devise, la durée et les catégories d’un budget.

### Budgets partagés

Pour les budgets partagés, l’utilisateur peut inviter d’autres utilisateurs (via leur nom d’utilisateur) à rejoindre le budget. Les personnes concernées reçoivent une notification pour accepter l’invitation.

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.

## Fonctionnalités optionnelles

### Vue globale



#### Diagramme en barres

Le diagramme en barres permet la comparaison entre les dépenses et les revenus d’argent par mois et par année sous forme de deux barres de couleurs différentes.

### Liste de souhaits

L’utilisateur peut créer une liste de souhaits, composée de différents articles qu’il aimerait prochainement acheter. Ces articles sont identifiés par un nom, éventuellement un lien internet et un prix.

Cette liste est mise à jour en fonction du budget actuel. Si l’épargne de l’utilisateur est suffisante, alors le produit est catégorisé comme étant achetable. Si au contraire l’utilisateur n’a pas suffisamment de fonds disponibles, une estimation de temps d’attente avant achat possible est calculée sur la base de l’évolution de l’épargne.

### Prévisions d’achat

Liste de souhaits “prioritaire”, qui correspond à “il faudrait (absolument) que j’achète l’article X avant la date Y”, X étant un objet (p.ex un cadeau, un outil, une fourniture, …).

La prévision d’achat est comme la liste de souhaits au niveau des informations, excepté qu’il faut une date limite pour l’achat. L’utilisateur a également la possibilité de mettre une alarme un certain temps avant la date limite, ainsi l’application pourra envoyer un rappel. La dépense sera effective une fois que l’utilisateur a confirmé l’achat.

### Budgets partagés

Depuis un budget partagé, il est possible de créer les dettes sous-jacentes concernant les utilisateurs du budget. Par exemple, si dans un budget partagé une personne paie l’entièreté d’un produit, il peut créer des dettes synchronisées avec chacun des autres utilisateurs, soit automatiquement (en divisant le montant de manière équitable entre les personnes) soit en précisant le montant pour chaque utilisateur.

### Exportation en PDF

Chaque vue est exportable au format PDF par l’utilisateur.

### Dettes

Possibilité de scanner les documents en lien avec la dette et les garder dans l’application avec la dette concernée. Fichier à titre informatif, l’application ne l’interprète pas.

### Simulation

Une simulation est possible en se basant sur les revenus et les dépenses récurrentes (salaire, abonnements, factures, loyer, capitalisation du compte, ...) ainsi que sur les données utilisateur déjà dans l’application, en prenant par exemple des moyennes de dépenses.

Un utilisateur peut modifier ses dépenses et revenus en mode simulation, s’il souhaite voir comment évolue son budget sans pour autant sauvegarder les modifications réalisées. L’activation de ce mode est repérée par des indications visuelles, permettant à l’utilisateur de ne pas se tromper de mode.

# Architecture (A METTRE A JOUR)



L’architecture logicielle est conçue pour fonctionner sur le système d’exploitation Windows 10. Le logiciel est découpé en trois couches, chaque couche a une responsabilité précise :

* La couche présentation en bleu permet de présenter les informations à l’utilisateur dans des fenêtres graphiques.
* La couche logique métier en magenta gère toute la logique métier de l’application.
* La couche d’accès aux données en rouge va permettre de gérer la persistance et l’accès aux données.

# Choix d’implémentation

## Langage de programmation et GUI

L’application a été implémentée en Java. En ce qui concerne les aspects graphiques, nous avons utilisé JavaFX, pour pouvoir séparer de manière claire la couche purement graphique de celle qui implémente les fonctionnalités graphiques, comme les boutons.

## Framework

Nous avons également utilisé 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.

# 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 réel 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 par en 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 peut être récurrent ou non. S’il ne l’est pas, il faut spécifier une date de début et une date de fin. En revanche s’il l’est, en plus des deux dates précédentes, il faut rajouter un intervalle de date (ex : chaque deux semaines). La notion de récurrence d’un budget sera mise à jour à l’aide d’un trigger event sur la base de données.

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. Elles peuvent également être récurrentes, si cela a été spécifié lors de leur création. Un fois la date de la récurrence atteinte, la transaction associée est dupliquée et la prochaine date de récurrence est recalculée. Cela est également mis en place avec un trigger event sur la base de données.

## 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 décliné.

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.

# 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

Dans l’arborescence du projet nous avons créé un package DAL (data access layer). Cette couche va nous permettre d’interagir avec une base de données centralisée PostgreSQL et une base de données embaquée Derby.



DAL 12 - Arborescence des packages

### 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.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.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.irepositories

Ce package contient toutes les interfaces qui exposent les méthodes d’accès aux données, dont les opérations CRUD par exemple. L’implémentation de ces répertoires se fait une fois pour Derby et une fois pour PostgreSQL, bien que le code reste assez semblable.

### 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.

Plus d’informations suivront dans cette partie car pour le moment nous n’avons pas encore terminé.

# 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.

## 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 contiendra plusieurs éléments, nous utilisons le fichier LoginRegister.fxml. Le conteneur principal de celle-ci sera 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
* 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 (à voir si on le met). 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 temps (depuis le début ou 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.

### Vue création/modification

## Fenêtre des listes de catégories



GUI 6 - Aperçu de la liste des catégories dans SceneBuilder

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 TilePane 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).

### Contrôleur

Pas encore implémenté

## Fenêtre de la liste des transactions



GUI 7 - Aperçu de la liste des transactions dans SceneBuilder

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 JFXListCell qui permet de lister nos transactions. Au fond nous avons un ChoiceBox qui est le bouton ajouter. Au clic nous devons choisir entre une dépense ou un revenu. Ce choix n'est pas définitif, nous étudions la possibilité d'utiliser un JFXNodeListe qui correspond plus à ce que nous pourrions avoir besoin.

### Contrôleur

Pas encore implémenté

## Problèmes rencontrés

### 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.

# Logique métier (BLL)

## Introduction

## Classe Authentication

# Conclusion