UNIVERSITÉ LIBRE DE BRUXELLES

ULB

Faculté de Lettres, Traduction et Communication

Conception et gestion de banques de données : Rapport de projet

Budget Squirrel : Application de gestion de budget

Milena ALBU
Marie HUYSMAN

Rapport de projet écrit dans le cadre du cours STIC-B505 : Conception et gestion de banques de données

Année académique 2020–2021

Table des matières

1 Introduction

Dans le cadre de ce projet, nous avons choisi de développer une application appartenant au domaine de la gestion financière et permettant la gestion de budget à une échelle personnelle. Le rapport ci-dessous documente le processus de conception, ainsi que les développements successifs et les améliorations apportées à l'idée originale décrite dans le rapport précédent.

Après une brève présentation du domaine d'application, nous présenterons notre schéma conceptuel, ainsi que sa traduction en schéma relationnel. Ensuite, nous décrirons les différents éléments des scripts SQL que nous avons mis en place : clés uniques et étrangères, contraintes au niveau de la base des données, checks et triggers, ainsi que privilèges d'accès. Cette description sera suivie d'une brève présentation de l'application Web que nous avons développée. Nous terminerons ce rapport en offrant quelques informations sur le développement du projet, ainsi qu'en exposant les conclusions que nous avons tirées lors de la finalisation de ce dernier.

En annexe à ce rapport, vous trouverez deux fichiers sql : `initdb.sql`, qui contient le script permettant la création de la base de données en elle-même, et `basicdata.sql`, qui peuple la base de données de quelques informations, permettant ainsi de parcourir les différentes pages de l'application ainsi que la base de données afin d'en comprendre leur fonctionnement. Toujours en annexe à ce rapport, nous avons joint les pages de notre application (9 en tout) : `index.html`, `inscription.php`, `connexion.php`, `homepage.php`, `profil.php`, `historique.php`, `enregistrement.php`, `stat.php`, et `logout.php`. S'y trouvent également un dossier css contenant le boilerplate CSS Skeleton [1] ainsi que les modifications que nous y avons apporté, un dossier img contenant les images utilisées dans l'application, et un fichier `utils.js` contenant une fonction JavaScript de redirection de page.

NOTE : pour le zip final à rendre, ne pas inclure le readme, ni le dossier du rapport latex.

1.1 Domaine d'application et utilisateurs cibles

Une application de gestion de budget est une application qui permet, au minimum, l'enregistrement ainsi que la suppression des transactions dans une base des données, la gestion des utilisateurs et de leur budget, et la possibilité d'agréger une vue d'ensemble sur les transactions financières enregistrées. Afin de mettre en œuvre une telle application, il est nécessaire de connaître les besoins minimaux des utilisateurs potentiels, ainsi que les différentes techniques de programmation à employer pour répondre de manière pertinente, voir optimale, à ces besoins.

1. `http://getskeleton.com/`

Dans le cas de Budget Squirrel, l'application se focalise sur les utilisateurs qui n'ont pas nécessairement une expertise spécifique en ce qui concerne planning financier, mais qui souhaitent néanmoins avoir une vue d'ensemble, ainsi qu'un suivi quotidien, de leurs dépenses et leurs revenus, afin de mieux les analyser de manière rétrospective, mais aussi prévisionnelle. L'application est à usage restreint et, dans sa version actuelle, c'est-à-dire en usage libre, lié aux profils de ses utilisateurs, mais sans être protégée par un système de mot de passe, nous recommandons qu'elle soit employée au sein d'une réseau domestique, et sur un serveur local.

1.2 Cas d'utilisation du système

Comme convenu dans le rapport précédent, l'application sait gérer plusieurs cas d'utilisation : connexion et déconnexion, création et modification de profil, enregistrement et suppression des transactions, vue historique propre à chaque utilisateur, et accessible en mode consultation ainsi qu'en mode édition (permet la suppression des transactions listées), et vue statistique avec visualisation dynamique des données liées aux transactions par mois (graphique en barres), et par catégorie (diagramme circulaire).

L'application Budget Squirrel compte 9 écrans, divisés en 4 catégories :

1. gestion de profil (écran d'accueil index.html, page d'accueil de l'application, connexion, inscription, profil, et log out) ;
2. enregistrement (écran d'enregistrement de transaction) ;
3. historique des transactions (écran historique), et
4. statistiques (écran statistiques).

Dans la réalisation d'écrans, des éléments de PHP (76.2 %), CSS (6.4 %), HTML (1.9 %), et JavaScript (0.1%) ont été utilisés. Le rapport, quant a lui, a été rédigé en utilisant LaTeX (15.4 % de notre projet).

Les points suivants présentent succinctement les cas d'utilisation que nous avons mis en place dans le cadre de ce projet.

1.2.1 Première connexion au système

1.2.2 Connexion et déconnexion

1.2.3 Gestion du profil

1.2.4 Enregistrement de transactions financières

1.2.5 Consultation de son historique personnel de transactions et suppression de transactions

1.2.6 Consultation de statistiques

2 Schéma conceptuel

Le schéma conceptuel de la base des données Budget Squirrel reste conforme au schéma validé lors de la première étape du projet. Nous comptons 5 tables principales, ainsi que 3 tables issues de la matérialisation de l'héritage sur la table transaction financière.

En ce qui concerne les attributs, chaque table contient au minimum un identifiant (clé primaire), ainsi que des contraintes (clé étrangère, ou bien contraintes d'unicité, selon les besoins).

Nous avons appliqué quelques corrections par rapport au schéma conceptuel de notre premier rapport. La structure générale reste très proche de celle de notre premier schéma, mais nous avons apporté des modifications aux tables catégorie_tf, virement, budget_mensuel, et carte, soit pour nous conformer aux corrections demandées après la remise du premier rapport, soit après discussion lors d'une des guidances, lorsque nous nous sommes rendues compte que certains de nos éléments rajoutaient une complexité inutile à l'utilisation de l'application.

La table carte est devenue une entité faible, car nous considérons qu'une carte ne peut pas être identifiée seulement par son numéro, mais par la combinaison entre son numéro et le NISS de l'utilisateur. De plus, pour permettre une « suppression »des cartes par l'utilisateur (ou plutôt une désactivation), nous avons ajouté un attribut permettant de définir le statut (activé ou non) de la carte. Enfin, pour ne pas afficher le numéro de la carte à travers l'application, nous avons ajouté un attribut de nom, qui permet à l'utilisateur de nommer sa carte.

Nous avons aussi ajouté un élément de description à la table catégorie, ce qui permet de mieux comprendre chaque élément de la table, autant dans la base de données que du côté de l'application.

Des contraintes concernant les dates ont également été ajoutées, car il n'est pas possible qu'une date de transaction (et par conséquent, le mois et l'année d'un budget) précède la date de naissance de l'utilisateur qui l'a créée.

Une communication n'étant pas obligatoire lors d'un virement, nous considérons dans cette nouvelle version du schéma que l'attribut communication de la table virement est un attribut facultatif.

Enfin, en cours de développement, nous avons fait le choix de supprimer les contraintes de reste et de clôture de chaque budget mensuel, car cela pouvait poser des problèmes de calcul : si le budget du mois de février est clos avant celui de janvier, et que son reste est ajouté au mois de mars, par exemple, à la clôture du budget du mois de janvier, il faudrait ajouter le reste au mois de février, et répercuter le changement sur le mois de mars (et ainsi de suite). Pour proposer une application plus flexible, nous avons fait le choix de simplifier la table budget_mensuel et son

fonctionnement, en supprimant les attributs de reste et de statut. La clôture d'un budget n'est donc pas possible. Cette suppression reste mineure, car l'application se concentre principalement sur la gestion du budget et des différentes transactions, que se soit de manière rétrospective, ou de manière prédictive. L'objectif principal reste atteint, et la gestion des différents budget n'en est que plus souple pour l'utilisateur.

SCHEMA : ajouter contrainte (t,e) de l'héritage + faire dernières régularisations par rapport au reste du contenu

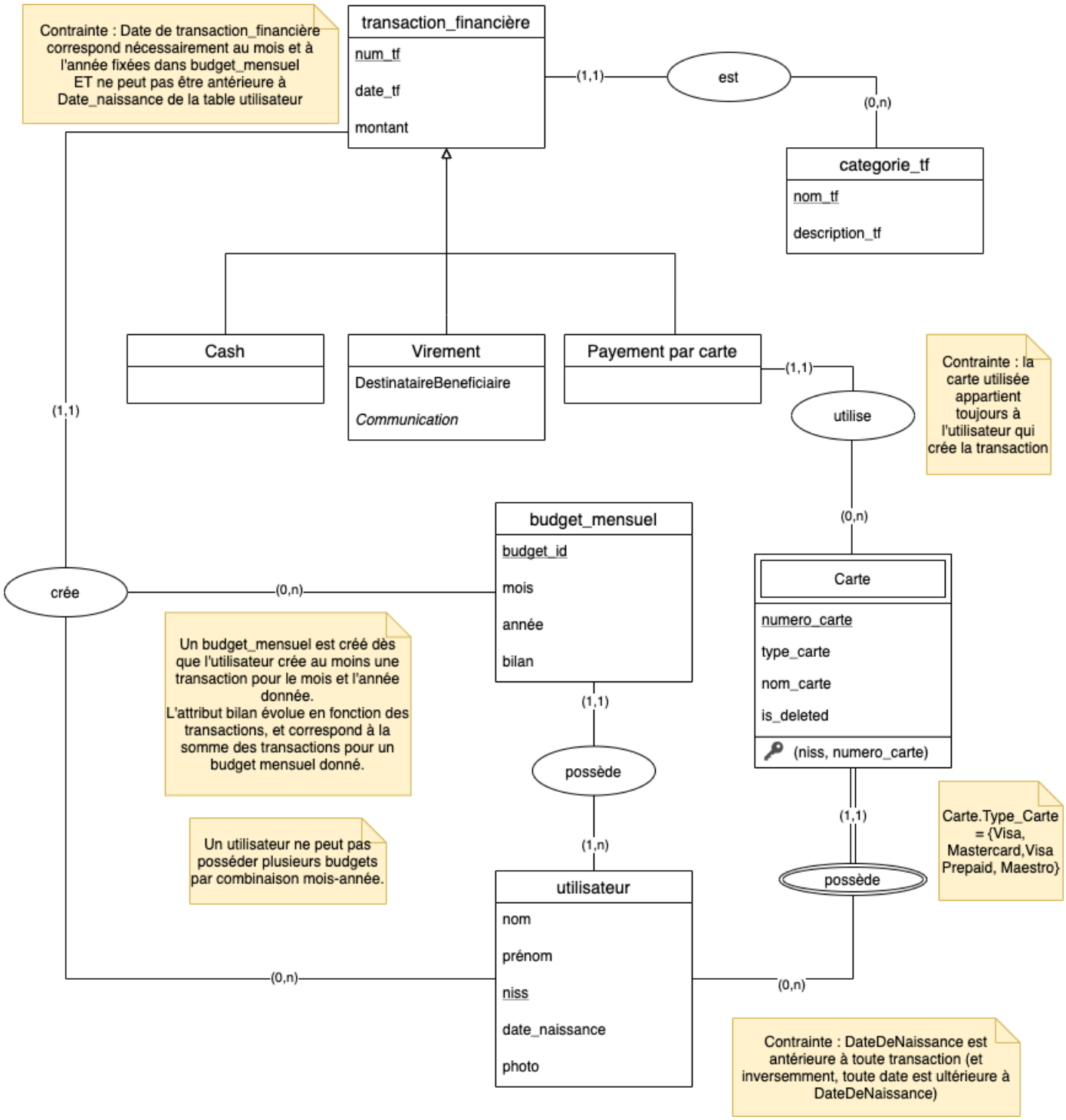

FIGURE 1 – Schéma conceptuel entité-association de la base de données exploitée par Budget Squirrel

3 Schéma logique relationnel

Vous trouverez ci-dessous notre traduction du schéma entité association présenté à la section précédente, ainsi que les contraintes supplémentaires, et quelques explications concernant nos choix de traduction.

utilisateur(nom, prenom, niss, date_naissance, photo)
        utilisateur.niss = {froggy.png, gollum.jpg, politecat.jpg, raccoon.jpg}

carte(nom_carte, numero_carte, type_carte, niss_util, is_deleted)
        carte.niss_util référence utilisateur.niss
        carte.type_Carte = {Visa, Mastercard,Visa Prepaid, Maestro}

budget_mensuel(budget_id, mois, annee, bilan, niss_util)
        budget_mensuel.niss_util référence utilisateur.niss

transaction_financiere(num_tf, date_tf, montant, budget_id, niss_util, cat_tf)
        transaction_financiere.budget_id référence budget_mensuel.budget_id
        transaction_financiere.cat_tf référence categorie_tf.nom_tf
        transaction_fianciere.niss_util référence utilisateur.niss

tf_cash(num_tf)
        tfcash.num_tf référence transaction_financiere.num_tf

tf_virement(num_tf, destbenef, *communication)*
        tfvirement.num_tf référence transaction_financiere.num_tf

tf_carte(num_tf, numero_carte)
        tf_carte.num_tf référence transaction_financiere.num_tf
        numero_carte référence carte.numero_carte

categorie_tf(nom_tf, description_tf)

Voici les contraintes que nous avons définies dans cette traduction, afin qu'elle corresponde à notre schéma entité-association :

- utilisateur.niss ne peut contenir que 11 caractères, spécifiquement 11 chiffres, pas plus, ni moins ;
- La valeur par défaut de carte.is_deleted est 0 (zéro) ;
- La valeur de carte.numero_carte contient uniquement des chiffres, et ne peut contenir que 16 ou 17 chiffres ;
- budget_mensuel.budget_id correspond a une combinaison unique de niss_util, mois et année, ainsi, il est impossible pour un même utilisateur de posséder plusieurs budgets pour le même mois et la même année ;
- budget_mensuel.bilan correspond à la somme de toutes les transactions pour un utilisateur, un mois et une année données ;

— transaction_financiere.date_tf (et par extension, le mois et l'année de budget_mensuel) sont nécessairement ultérieures à utilisateur.date_naissance ;

— transaction_financiere.budget_id ne peut référencer qu'un budget_id dont le mois et l'année correspondent à transaction_financiere.date_tf ;

— Dans la table tf_carte, transaction_financière.num_tf et carte.numero_carte référencent obligatoirement le même niss_util : un utilisateur ne peut qu'utiliser les cartes qui lui sont liées pour les transaction qui lui sont liées ;

— transaction_financière.num_tf doit se retrouver une et une seule fois soit dans la table tf_carte, soit dans la table tf_virement, soit dans la table tf_cash ; **Côté DB, on ne vérifiait pas, j'ai donc ajouté 3 trigger qui le font (1 pour chaque table)**

— **Autres ?**

L'héritage entre la table transaction_financière et les tables cash, virement, et payement par carte a été traduit par une matérialisation, que nous représentons graphiquement à la figure 2. Nous avons choisi d'utiliser la matérialisation afin de conserver la super-entité, ainsi que les sous-entités, et la relation sémantique qu'elles entretiennent. La relation est considérée comme totale et exclusive : une transaction financière est au moins une transaction en liquide, par virement ou payée par carte, mais ne peut pas être de plus d'un type à la fois.

L'association un à plusieurs entre utilisateur et budget_mensuel a été traduite en plaçant la référence du côté "un" de l'association : ainsi, on retrouve utilisateur.niss dans budget_mensuel.niss_util. Il en est de même pour la table transaction_financière, à laquelle nous avons ajouté une référence et au budget_mensuel, et au NISS de l'utilisateur. Toujours en suivant la même logique, l'entité faible carte contient son attribut d'association identifiante, lié au NISS de l'utilisateur, et la clé de la table est formée de la combinaison NISS - numéro de carte ; la table transaction_financiere contient l'attribut qui référence categorie_tf.nom_tf ; et tf_carte contient les références à transaction_financière.num_tf et à carte.numero_carte.

A CONFIRMER : Nous n'avons pas dû traduire l'association n-aire entre transaction_financiere, budget_mensuel et utilisateur, tout simplement car elle n'est pas nécessaire : la table transaction_financiere contient déjà toutes les informations de l'association, à savoir l'association num_tf - budget_id - niss. Une table séparée représentant le lien entre les trois tables aurait été nécessaire si, par exemple, des données étaient directement liée à l'association qui les lie.

AUTRES JUSTIFICATIONS À AJOUTER ?

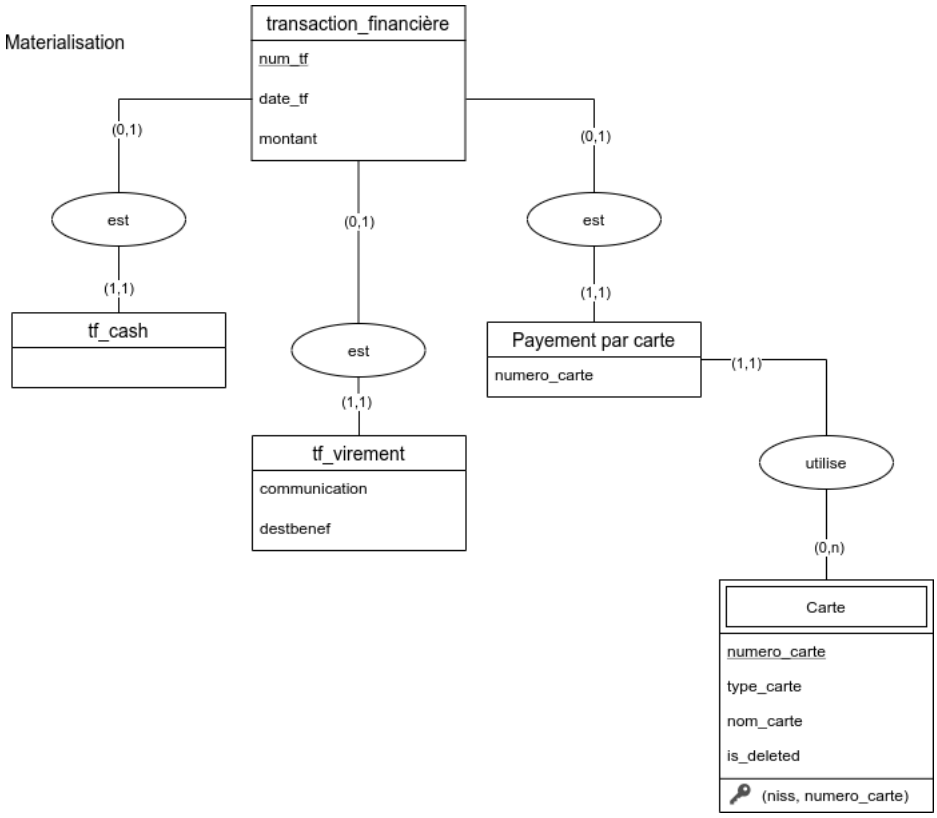

FIGURE 2 – Matérialisation de l'héritage entre les transactions

4 Code SQL

Dans cette section, nous présentons les différents éléments de SQL qui sont présents dans notre projet, ainsi que leur utilité respective. Comme décrit dans notre introduction, vous trouverez en annexe deux fichiers SQL : `initdb.sql`, et `basicdata.sql`. Le premier contient le script de création des tables, la mise en place des déclencheurs, la création des vues, et enfin la définition des utilisateurs et de leurs privilèges respectifs. Le second peuple les différentes tables, en ajoutant : 3 utilisateurs, 2 ou 3 cartes par utilisateur, les catégories de transaction et leur description, et quelques transactions pour chaque utilisateurs.

Ainsi, une fois les deux scripts SQL exécutés, l'application peut être utilisée et des données d'exemple sont déjà disponibles.

4.1 Description du code de création de la base de données

Après une requête de suppression de la base de données budgetsquirrel si elle existe déjà, une nouvelle base de données du même nom est créée et appelée à être utilisée. Les premières requêtes sont des `CREATE TABLE`, et permettent de créer les tables qui sont nécessaires à l'utilisation de l'application. Elles correspondent

à notre schéma logique relationnel. Ainsi, le script SQL crée les tables suivantes : `utilisateur`, `carte`, `budget_mensuel`, `categorie_tf`, `transaction_financiere`, `tf_cash`, `tf_virement`, et `tf_carte`.

Les contraintes prévues par le schéma logique relationnel pouvant être traduites directement dans la création des tables sont alors établies. Tous les attributs obligatoires sont déclarés comme non nuls et la longueur de utilisateur.niss est fixée à 11, la longueur de carte.numero_carte est fixée entre 16 et 17, grâce à des `CHECK` que nous décrivons plus loin. Toutes les clés primaires et étrangères sont également déclarées dans cette partie du script. Lorsque cela est nécessaire, nous ajoutons des contraintes `UNIQUE`, toujours pour respecter notre schéma : c'est le cas pour la contrainte `carte.uc_carte`, `utilisateur.uk_utilisateur`, et `uc_budget_mensuel`.

Dans la table `budget_mensuel` comme dans la table `transaction_financiere`, nous utilisons la commande `AUTO_INCREMENT` afin d'établir que les clés primaires de ces tables, à savoir `budget_mensuel.budget_id` et `transaction_financiere.num_tf`, sont des colonnes numériques qui sont incrémentées automatiquement à chaque nouvelle insertion.

Nous avons également ajouté quelques contraintes liées à la notre application. Par exemple, nous donnons une valeur par défaut à l'url de la photo (`utilisateur.photo`), et nous établissons que les seules valeurs acceptées par l'attribut utilisateur.photo sont celles correspondant aux photos de notre dossier d'images. De manière similaire, nous établissons que les seules valeurs acceptées pour `carte.type_carte` sont Visa, Mastercard, Visa Prepaid, et Maestro. De plus, une carte créée est par défaut considérée comme non supprimée, ainsi, carte.is_deleted a une valeur par défaut de 0.

Puisque l'utilisateur peut supprimer ses transactions, nous avons ajouté des `ON DELETE CASCADE` aux `num_tf` référencés dans les tables `tf_cash`, `tf_virement`, et `tf_carte`, afin que la transaction soit supprimée des tables qui contiennent des informations sur son type. Un administrateur pourrait éventuellement vouloir supprimer totalement un utilisateur de la base de données, lui et toutes ses cartes, transactions, ou encore changer certaines informations liées à ce dernier, et que cela se répercute sur le reste de la base de données. Afin de gérer ces deux situations, nous avons ajouté des `ON DELETE CASCADE` et des `ON UPDATE CASCADE` à chaque référence du NISS de l'utilisateur en tant que clé étrangère.

TERMINER CET AJOUT de delete et update cascade, j'ai l'impression qu'il en manque dans initdb.sql

Autres infos à rajouter ?

Après la création des tables que nous venons de présenter, `initdb.sql` peut être divisé en X grandes parties : la création des triggers/déclencheurs, la création des vues, et la gestion des privilèges. Toutes ces parties sont décrites dans les points suivants.

4.2 Éléments de SQL avancé

Vous trouverez dans les points suivants la description de divers éléments de SQL avancé auxquels nous avons fait appel dans le cadre de notre projet.

4.2.1 Contraintes garanties par des check

Comme décrit précédemment, nous faisons appel à des `CHECK` afin de vérifier la validité de certaines valeurs d'attributs dans notre base de données. Il y a en tout 5 checks : `chk_niss`, qui vérifie que le NISS enregistré fait bien la bonne longueur (exactement 11 chiffres) `chk_photo`, qui vérifie que la valeur de la photo est bien une de celles que nous avons choisie, `chk_numero_carte_low` et `chk_numero_carte_high`, qui vérifient que le numéro de carte entré correspond bien à la longueur demandée (entre 16 et 17 chiffres), et enfin, `chk_type_carte`, qui vérifie que `type_carte` correspond bien aux différents types de cartes définis précédemment.

4.2.2 Requêtes de consultation

4.2.3 Requêtes de mise à jour

4.2.4 Vues utilisées

La base des données contient quatre vues qui sont utilisées dans la construction d'historiques et des statistiques dans l'application :

1. historique_v, qui liste chronologiquement les informations liés aux transactions effectuées par un utilisateur,
2. stat_depenses_revenus_mois, qui liste chronologiquement les dépenses et les revenus enregistrés par l'utilisateur, en faisant également le bilan des dépenses, le bilan des revenus, et le bilan total,
3. stat_cat, qui liste la répartition totale des dépenses et des revenus par catégorie et par utilisateur,
4. et stat_types, qui liste la répartition du budget global par type de transaction employé dans le payement, ainsi que par nombre d'emplois de chaque type de transaction, pour un bilan global des dépenses et revenus.

Ces vues sont utiles à la gestion et à l'agrégation des données, et permettent notamment d'offrir à l'utilisateur de l'application différents graphiques et informations agrégées sur son budget.

Ci-dessous, vous trouverez une description détaillée de leur fonctionnement.

Historique

Statistiques mensuelles

Statistiques par catégorie

Statistiques par type de transaction

4.2.5 Déclencheurs

Afin de conserver toutes les contraintes définies dans notre schéma, nous avons défini 7 déclencheurs : `trg_before_ajout_tf`, `trg_after_ajout_tf`, `trg_after_suppr_tf`, `trg_before_ajout_tf_carte`, `trg_before_ajout_tf_carte`, `trg_before_ajout_tf_cash`, et `trg_before_ajout_tf_virement`. La création de ces déclencheurs suit directement la création des tables, et permet, entre autres, d'établir les contraintes les plus complexes (qui incluent plusieurs tables). Nous expliquons le fonctionnement de chacun des ces déclencheurs dans les points suivants.

Vérifier les informations avant l'ajout d'une transaction financière
Le déclencheur `trg_before_ajout_tf`, qui se fait avant chaque insertion sur la table financière, permet de vérifier plusieurs informations. Tout d'abord, une vérification sur la date est faite : si la date de transaction est antérieure à la date de naissance de l'utilisateur, un message d'erreur est renvoyé et la transaction de s'enregistre pas. Ensuite, l'objectif est de trouver le `budget_id` à lier à la transaction financière à créer. Pour ce faire, nous vérifions d'abord, via un `SELECT`, s'il existe déjà un `budget_mensuel` correspondant à la combinaison de mois, d'année et de NISS. Si ce n'est pas le cas (que le résultat `COUNT(*) = 0`), le `budget_mensuel` correspondant est créé. Ensuite, dans tous les cas, le `budget_id` du `budget_mensuel` correspondant est sélectionné, et défini comme à écrire dans l'insertion de la nouvelle `transaction_financiere`.

Écrire (et réécrire) le bilan du budget mensuel en cas d'insertion ou de suppression dans la table des transactions financières
Deux déclencheurs sont utilisés pour que `budget_mensuel.bilan` corresponde toujours à la bonne somme de transactions : `trg_after_ajout_tf` et `trg_after_suppr_tf`. Dans les deux cas, après une insertion ou une suppression de la table `transaction_financiere`, le résultat `bilan_total_mois` de la vue`stat_depenses_revenus_mois` correspondant au `budget_id` et au

niss_utilisateur de l'insertion ou de la suppression est sélectionné, et permet de faire une modification de budget_mensuel.bilan.

Vérifier que la carte utilisée lors de la transaction appartient bien à l'utilisateur
Lors de l'ajout d'une transaction dans la table tf_carte, il est important, afin de respecter une des contraintes que nous avions définies, de vérifier que la carte qui est utilisée appartient bien à l'utilisateur qui a fait la transaction. Le déclencheur trg_before_ajout_tf_carte, avant l'insertion dans la table enregistrant les transactions par carte, va donc récupérer le NISS présent dans la table des transactions financières créées qui correspond au bon numéro de transaction, et le NISS correspondant à la carte utilisée. Si les deux NISS récupérés diffèrent, un message d'erreur s'affiche, et l'enregistrement ne se fait pas dans la table tf_carte.

Conserver la contrainte de généralisation totale et exclusive
Afin d'assurer que les transactions financières ne sont pas écrites plusieurs fois, mais bien une seule, dans une des trois tables qui définissent leur type (tf_carte, tf_virement et tf_cash), nous utilisons trois triggers : trg_before_ajout_tf_carte, trg_before_ajout_tf_cash, et trg_before_ajout_tf_virement. Chacun de ces déclencheurs fait la même vérification avant ajout. Grâce à des SELECT dans les deux autres tables, s'il existe déjà une ligne qui correspond au numéro de transaction, l'ajout dans la table de type ne se fait pas, et un message apparaît.

4.2.6 Droits d'accès

Enfin, le script initdb.sql définit également deux utilisateurs principaux, ainsi que leurs privilèges. Nous y définissons deux utilisateurs : utilisateur_app, qui est celui qui est appelé tout au long de l'application, et utilisateur_admin_db, qui, comme son nom l'indique, est l'utilisateur-administrateur. Alors que ce dernier a un accès illimité à la base de données (défini grâce à GRANT ALL PRIVILEGES), utilisateur_app ne peut faire que les actions définies par le script.

Les actions définies sont les suivantes :

— L'insertion, la modification et la sélection de lignes pour la table utilisateur, ce qui lui permet de s'inscrire, de modifier ses informations, et de pouvoir avoir accès à la liste des utilisateurs (cf. écran de connexion) ainsi qu'à ses informations ;

— le droit de sélection et d'insertion sur la table transaction_financiere, ce qui lui permet de visualiser et d'insérer de nouvelles transactions financières ;

— le droit à la sélection et à l'insertion dans la table `budget_mensuel` (le droit à la sélection n'est pas réellement nécessaire, vu que le trigger `trg_before_ajout_tf` automatise l'écriture des budgets mensuels)

— le droit à la sélection, la modification et l'insertion pour la table carte, afin de pouvoir utiliser, supprimer/désactiver, et créer ses cartes ;

— le droit de sélection sur la table `categorie_tf`, afin de définir la catégorie d'une transaction, et pouvoir consulter la description d'une catégorie sélectionnée dans son tableau statistique ;

— le droit de sélection des différentes vues que nous avons créées (`stat_depenses_revenus_mois`, `historique_v`, `stat_cat` et `stat_types`), afin d'avoir accès aux parties de l'application qui rassemblent des statistiques sur ses transactions ;

— **Autres ?**

!! SUPPRIMER doublon GRANT INSERT on budgetsquirrel.utilisateur to app dans initdb.sql

5 Description de l'application Web

L'application web que nous avons développée a pour objectif de permettre aux utilisateurs de s'inscrire, d'enregistrer leurs transactions, de consulter leur historique par mois, ainsi que des statistiques diverses sur la répartition de leurs dépenses, de leur revenus, et un bilan total de leur budget. L'accès à la base de données est restreint : ce n'est pas un accès root, mais bien un accès spécifiquement défini pour les utilisateurs de l'application. Nous en avions fait la description détaillée à la section précédente.

L'application contient 9 pages : `index.html`, `inscription.php`, `connexion.php`, `homepage.php`, `profil.php`, `historique.php`, `enregistrement.php`, `stat.php`, et `logout.php`. Ci-dessous, nous en décrivons leur articulation.

À l'ouverture de l'application (`localhost/budgetsquirrel/index.html` ou simplement `localhost/budgetsquirrel/`), l'utilisateur a deux possibilités : se connecter, ou s'inscrire.

FIGURE 3 – Landing page pour Budget Squirrel

S'il choisit de s'inscrire, il est redirigé vers un formulaire d'inscription, `inscription.php` où il doit rentrer son nom, son prénom, son NISS, et choisir une photo de profil. Si toutes les informations ne sont pas complétées, l'utilisateur est notifié du fait qu'il doit rentrer toutes les informations pour pouvoir s'inscrire.

Si un utilisateur est déjà inscrit sous le NISS entré, il en est notifié (et a la possibilité de se rediriger vers la page de connexion). Si le NISS n'est pas encore utilisé par un utilisateur, l'enregistrement se fait, l'utilisateur est notifié du succès de ce dernier, et il peut se rediriger vers la page de connexion. La vérification de la présence d'un utilisateur utilisant déjà le NISS se fait à deux niveaux : au niveau de la base de données en elle-même, comme nous l'avons vu au dessus, grâce à la contrainte de clé primaire, et également en PHP, en faisant une sélection de la table utilisateur (si la sélection par rapport au NISS entrée renvoie un résultat différent de zéro, l'utilisateur est notifié de

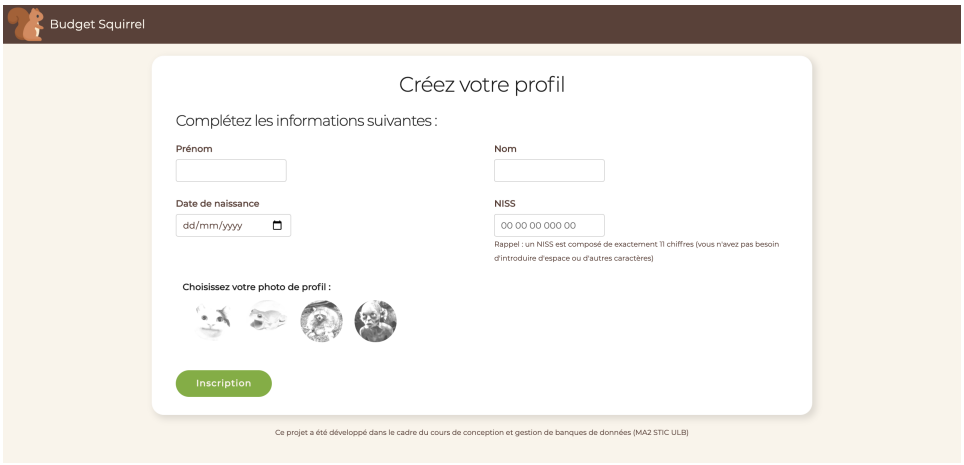

FIGURE 4 – Écran d'inscription - Budget Squirrel

l'impossibilité de créer un compte avec ce NISS).

La page de connexion `connexion.php`, par souci de simplification, est une simple sélection de profil : l'utilisateur a accès à la liste des utilisateurs, sélectionne son profil (nom et prénom) dans la liste, et se connecte. Ensuite, le NISS de l'utilisateur est récupéré par une méthode POST, enregistré par la méthode `SESSION` et appelé grâce à `session_start();` à chaque page de l'application où il est enregistré comme connecté.

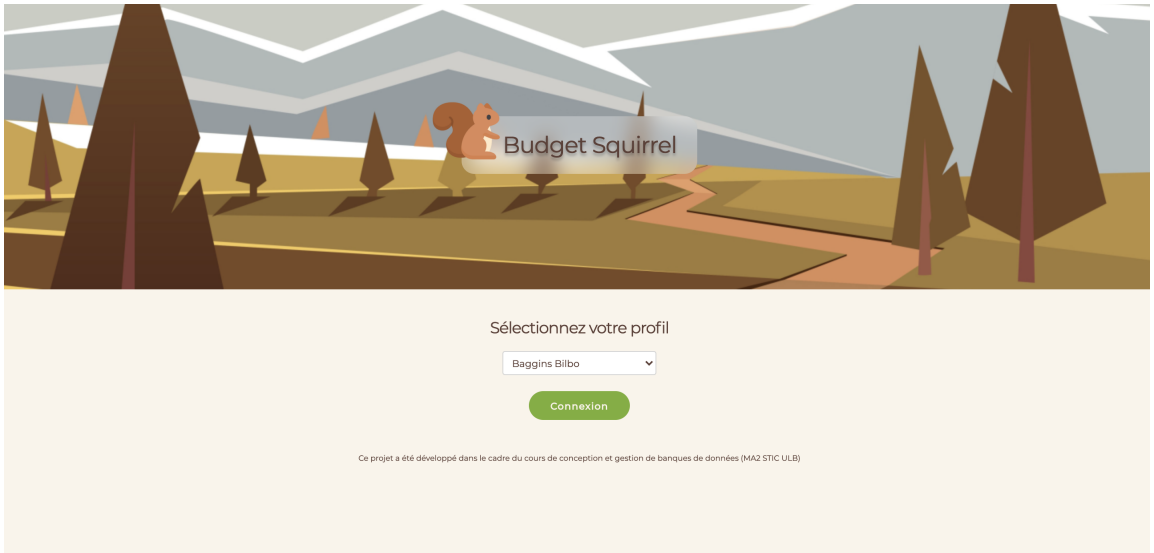

FIGURE 5 – Écran de connexion - Budget Squirrel

Une fois qu'il a sélectionné son profil et choisi de se connecter, l'utilisateur est redirigé vers une page d'accueil, `homepage.php`. La page d'accueil présente brièvement les différentes pages de l'application. Depuis la barre de navigation, visible sur toutes les pages (sauf celles ou l'utilisateur n'est pas considéré comme connecté), l'utilisateur

a accès : à la page d'accueil, à l'historique, à la page d'enregistrement, à la page de statistiques, et à sa page personnelle de profil. Il peut ainsi naviguer librement entre les différentes pages.

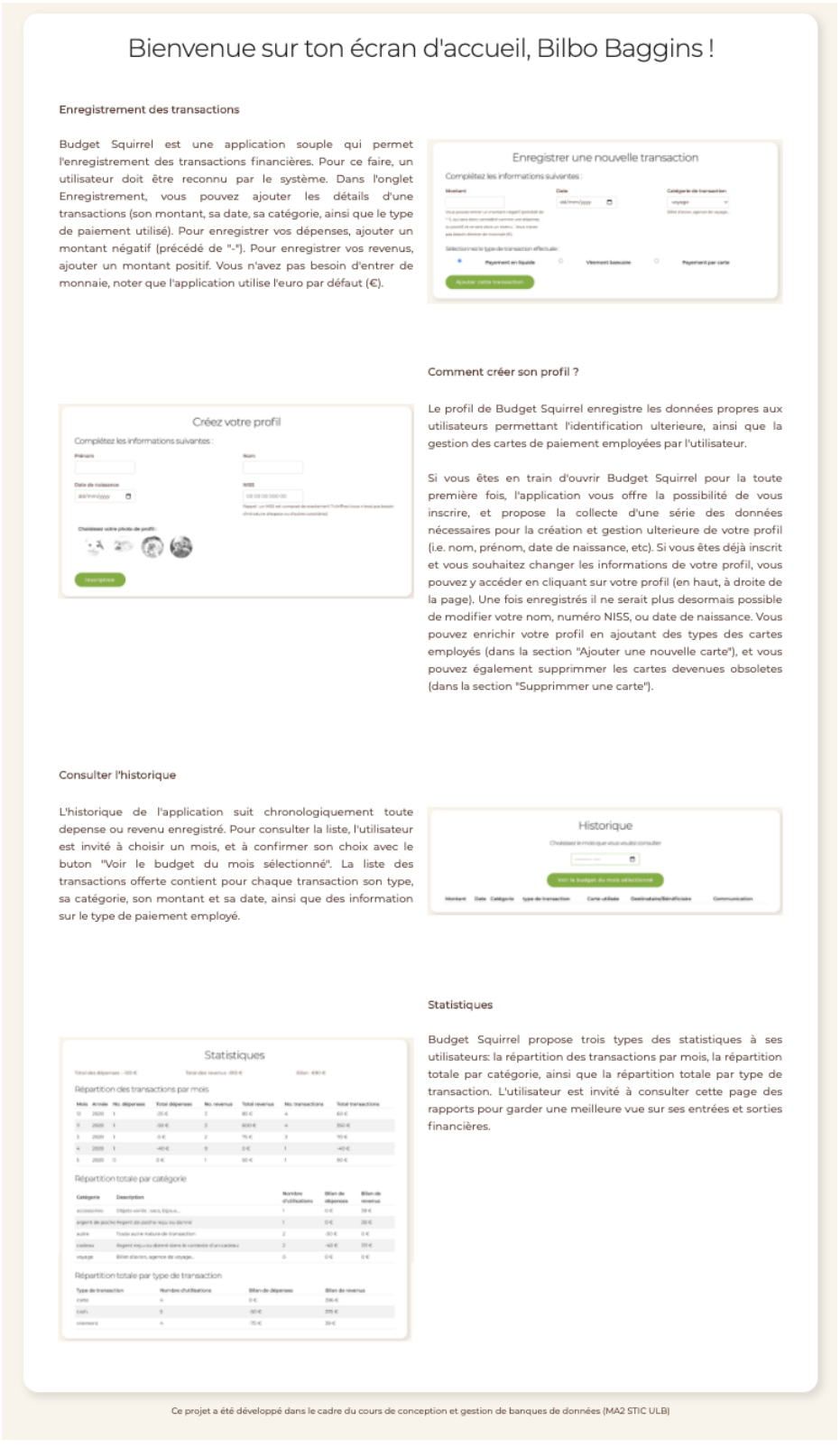

FIGURE 6 – Page d'accueil Budget Squirrel

La page personnelle de profil, `profil.php`, permet à l'utilisateur de visualiser ses informations : nom, prénom, NISS, date de naissance. Il peut également y ajouter et y supprimer ses cartes [2]. Les listes de ses cartes disponibles et de ses anciennes cartes sont également visibles depuis la page de profil. Il peut également y actualiser sa photo de profil.

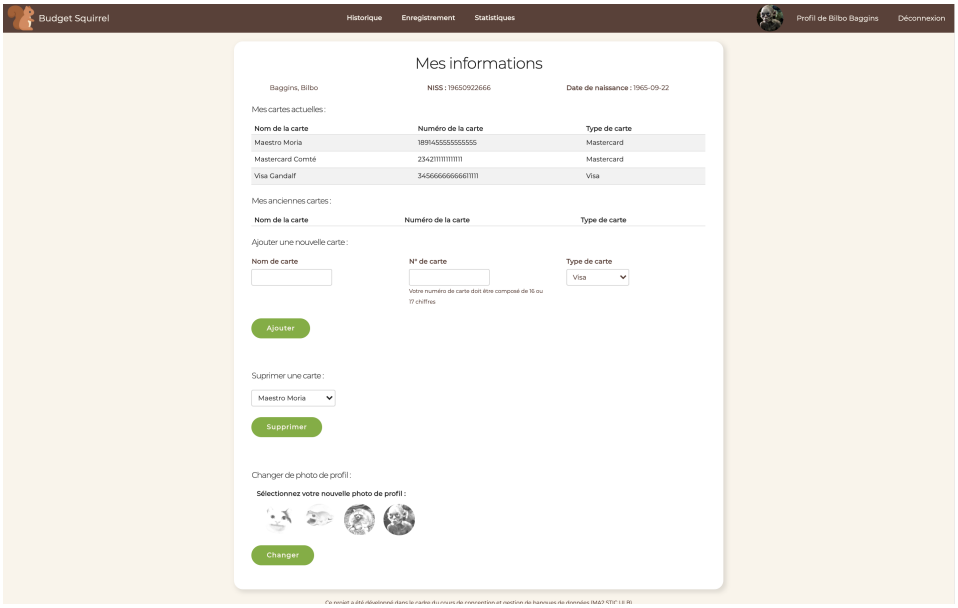

FIGURE 7 – Page personnelle Budget Squirrel

La page d'enregistrement, `enregistrement.php`, permet d'enregistrer des transactions. L'utilisateur est obligé d'entrer un montant, de sélectionner une date, une catégorie de transaction, et un type de transaction (à nouveau, la vérification se fait une fois en PHP, et une fois aussi du côté de la base de données).

En fonction du type de transaction effectuée, il doit également ajouter des informations supplémentaires, qui permettent de faire l'écriture dans la bonne table de type de transaction. L'utilisateur sélectionne, via un radio button, le type de transaction effectuée : soit un payement en liquide, soit un virement bancaire, soit un payement par carte (JavaScript nous permet de cacher une partie du formulaire en fonction du radio button sélectionné). Le payement en liquide ne demande aucune information supplémentaire, le virement bancaire demande d'introduire un destinataire ou un bénéficiaire, et éventuellement une communication, et un payement par carte nécessite de sélectionner la carte utilisée. Cet aspect est géré uniquement en PHP, et demande donc de faire deux opérations consécutives si l'on veut rentrer le type de transaction sans passer par l'application Web : d'abord, un `INSERT` dans la table

2. Cette suppression est, comme nous l'avions expliqué, une suppression logique du côté de la base de données : les cartes possèdent une colonne is_deleted , par défaut la valeur de la colonne est à 0, et si l'utilisateur « supprime »sa carte, la valeur passe à 1, et la carte est considérée comme virtuellement supprimée

`transaction_financiere`, puis un autre `INSERT` dans la table correspondant au bon type de transaction (c'est ce que nous faisons dans `basicdata.sql`).

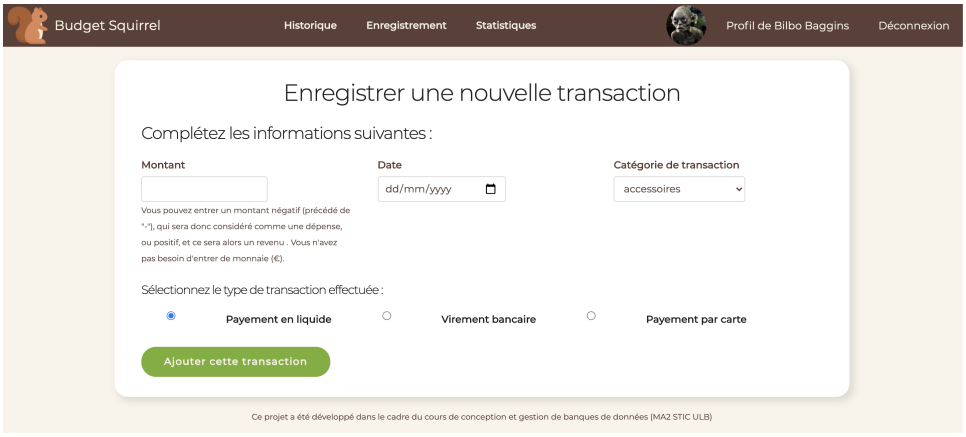

FIGURE 8 – Enregistrement des transactions en Budget Squirrel

La page historique, `historique.php`, permet à l'utilisateur de visualiser les transactions effectuées : pour cela, il doit d'abord sélectionner le mois et l'année dont il veut consulter le budget. Il a alors accès à un tableau affichant le montant, la date, la catégorie, et le type de transaction effectuée, et éventuellement la carte utilisée, le destinataire/bénéficiaire, et la communication. En dessous de la table, l'utilisateur peut également voir le bilan total du mois. Enfin, il peut également choisir de supprimer une des transactions affichées.

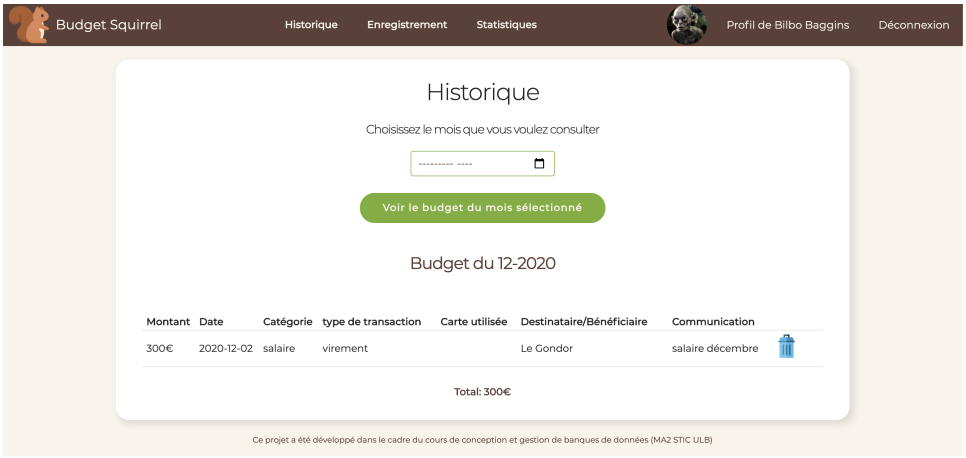

FIGURE 9 – Historique des transactions en Budget Squirrel

L'écran de statistiques, `stat.php`, offre différentes informations sur les données concernant l'utilisateur. Les informations premières informations offertes sont les suivantes : un total des dépenses, un total des revenus, et un bilan. Un tableau montrant la répartition des transactions (dépenses et revenus) par mois, ainsi que le bilan par mois, pour une vue plus condensée de l'historique, est également affiché, et accompagné de son graphique (généré en JavaScript) correspondant. Un autre tableau montrant la répartition totale par catégorie, couplé à une description de chaque catégorie, est aussi accompagné de son graphique, cette fois-ci un diagramme circulaire. Enfin, la répartition des transactions entre les différents types de payements (le nombre d'utilisations, le bilan des dépenses et le bilan des revenus pour chaque type de transaction) est affiché.

À tout moment, l'utilisateur peut également se déconnecter, en cliquant, en haut à droite de la barre de navigation, sur "déconnexion". Il est alors redirigé vers `logout.php`. Les informations de session (donc, le NISS sous lequel l'utilisateur est connecté) sont effacées, et l'utilisateur peut choisir de retourner à l'écran d'accueil pour se reconnecter ou s'inscrire.

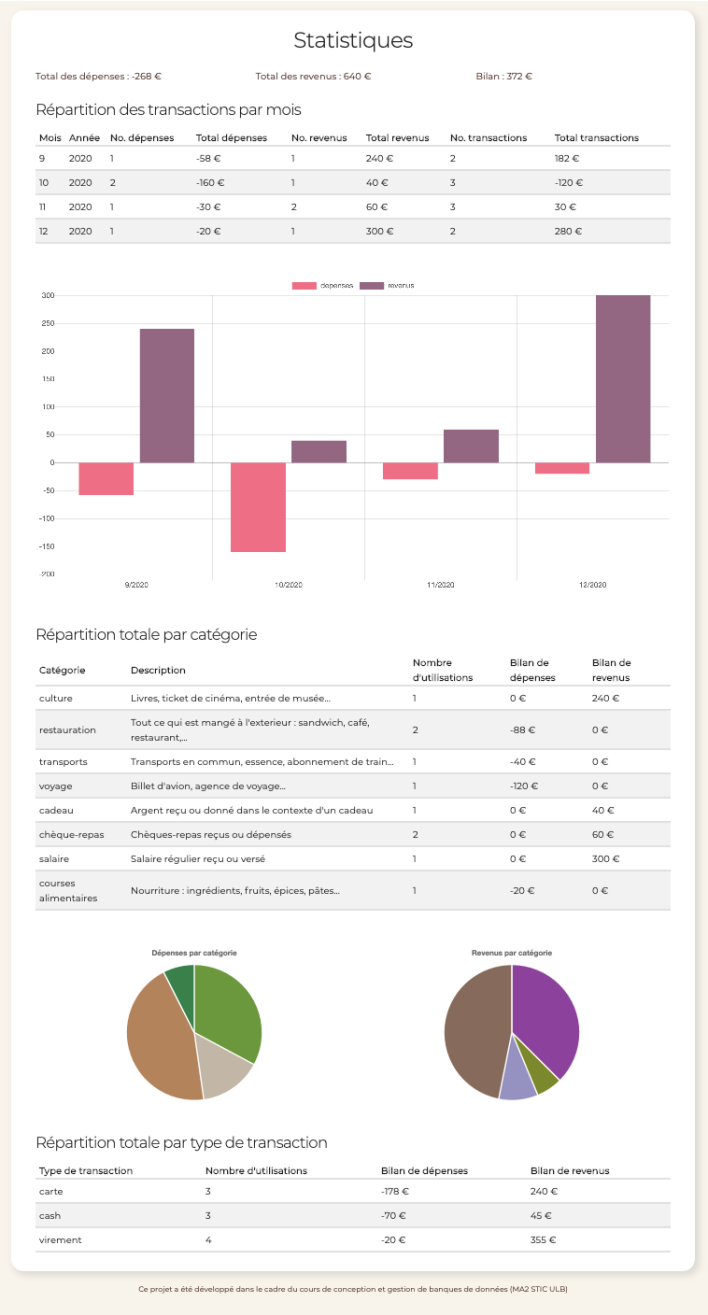

FIGURE 10 – Statistiques Budget Squirrel

6 Développement du projet

Budget Squirrel est une application constituée d'une base des données et d'une interface client. Le design, développement et administration d'une telle application nécessitent donc une chorégraphie particulière, reposant sur des livrables qui doivent être prêts à l'emploi dans une certaine séquence, et qui se trouvent la plupart de temps dans une symbiose étroite. De plus, certaines exigences du projet, établies ou agréées dès le départ, et révisés pendant les séances d'entretien projet, ainsi que cer-

taines contraintes liées à d'autres engagements impératives pour les développeurs se sont vite imposées comme des conditions demandaient une coordination ainsi qu'une communication ouverte et souple au sein du groupe. Le groupe s'est vite mis d'accord sur une division des tâches qui prend en compte les contraintes relatives aux disponibilités de chacune d'entre nous, ainsi qu'aux préférences des techniques et outils des développement, toute en gardant en vue l'impératif d'avoir touché au moins une fois à chaque écran et surtout d'avoir participé en tandem le plus que possible dans les parties couvrant la matière vue au cours.

6.1 Travail conceptuel et outils employés pour démarrer le projet

En suivant le calendrier communiqué au cours, le groupe à fait sa proposition de projet le 3 octobre, et a reçu une confirmation pour le deuxième sujet dans un bref délai.

Domaine application	Sujet	Utilisateurs cible	Description
Domotique	Gestion de stock de produits dans un frigo	Tout possesseur de frigo intelligent (types d'utilisateurs envisagés: administrateur et utilisateur simple.	**Entités**: frigo, produit, utilisateur. **Contraintes**: - un utilisateur a le droit de consulter les tables, - un administrateur a le droit de modifier les tables. Une **statistique** possible montrera le **taux de remplissage à une date precise, par type de produit** (par rapport à une quantite envisagé).
Planning financier	Système de gestion des dépenses	Personnes privées, banques (types d'utilisateurs envisagés: administrateur, banque, utilisateur simple).	**Entités**: utilisateurs (administrateurs, utilisateurs reguliers, banque), et dépenses (réparties par type de dépense), Une **statistique** possible montrera les dépenses **par type et dans une période de temps**.

FIGURE 11 – Sujets proposés en vue de développement

Une fois le choix de projet validé, nous avons commencé le travail de conceptualisation, avec un double-but : la conception d'un modèle entité-association, ainsi qu'une première ébauche d'interface client. Pour nous aider dans le démarche EA, nous avons employé l'application web draw.io, et en ce qui concerne l'interface client, l'outil Figma à été employé dans la construction des wireframes. Le résultat de ce travail s'est matérialisé dans un rapport préliminaire, présenté pour évaluation et validation le 31 octobre. C'est en suivant ce rapport préliminaire que nous avons par la suite développé les éléments décrits dans les sections 2 et 3 du présent rapport.

6.2 Design et techniques de travail

La partie design à été facilité par l'emploi d'une structure html ouverte, que nous avons taillé aux besoins de notre application. Cette étape à rajouté une première couche de complexité en ce qui concerne l'imbrication des langages utilisés, en ajoutant PHP et HTML aux requêtes SQL. Le PHP étant un langage très flexible, il nous a été relativement facile de créer de nombreuses contraintes, et de combiner différentes

requêtes SQL pour que l'application réponde exactement comme il faut à tout input de l'utilisateur. Par contre, un des défis que nous avons rencontré durant le développement de ce projet a été l'écriture de données en utilisant du SQL pur, sans l'aide de l'application. Certains éléments, facilement mis en place en PHP, étaient difficiles à traduire en SQL. Une deuxième couche de complexité à été rajouté par l'introduction des éléments javascript de visualisation dynamique des données (toggle screen, et graphes dynamiques) et des éléments en SQL d'aggregation des données (vues statistiques). De façon général, une des parties la plus difficile à été d'orchestrer, d'une façon robuste et cohérente, la partie d'insertions, mises à jour, et suppressions, afin d'éviter la perte des données, ou bien le double encodage. Faire le choix entre l'ajout des contraintes au niveau de client, ou bien au niveau de la base des données, ainsi que naviguer les contraintes implémentés, n'était pas toujours facile. Pour parer ces difficultés nous avons employé la technique de programmation en binôme (asynchrone, en utilisant GitHub pour partager les mises à jours au niveau du code, et Teams pour faciliter la communication). La taille restreinte du groupe à particulièrement facilité ce méthode de travail, et nous à permis une prise de décision rapide. La section 5 du rapport témoigne sur les choix qui ont été gardés et implémentés au niveau de design d'application.

6.3 Quelques scénarios de test

Nécessaire, si l'on donne déjà les cas d'utilisation du système?

7 Conclusion

7.1 Défis et solutions

7.2 Limites et possibilités de développement

Les points forts et faiblesses d'application, le temps nécessaire pour développer le concept avec deux personnes, les sources consultées pour la partie pratique (i.e. les slides, ou techniques vues au TP, ainsi que les conseils d'assistante, mais aussi w3schools ou bien stackoverflow). Finalement quelques mots sur l'expérience du projet.

Améliorations : - vérification du niss par rapport à la date de naissance de l'utilisateur - finalement, le bilan dans la table budget mensuel n'est pas obligatoire - choix de la matérialisation pour la traduction de l'héritage : remise en question