Rapport VeloMax

Base de données

Génération d’un utilisateur en lecture seule :

```
CREATE USER 'bozo'@'localhost' IDENTIFIED BY 'bozo';
GRANT SELECT ON VeloMax.* TO 'bozo'@'localhost';
FLUSH PRIVILEGES;
```

Relation avec la base de données

Pour lier la base de donnée SQL à notre base de données nous avons développé une classe DataBase qui agit comme une surcouche de MySQLConnection. Cette dernière se connecte à la base de données, peut exécuter des requêtes et les exporter sous la forme d’un QueryResult.

L’idée centrale de notre code était de développer des fonctions génériques pour gérer les opérations de consultation, modification, ajout etc... Grace à cela nous n’avons pas à écrire des requêtes sql pour chaque table et chaque table ajoutée dans notre base est automatiquement prise en charge par notre projet.

Classes principales

DataBase

- Se connecter à la base
- Vérifier l’existence de foreign_key
- Obtenir le noms de toutes les tables

Afin d’éviter les erreurs de mutualisation de ressources toutes les requêtes passent par des using afin de gérer les opérations d’ouverture et de fermeture de la base de données. on utilise aussi des try catch afin de ne pas faire crasher notre programme en cas d’erreur.

Table

- Sauvegarde local des tuples d'une table de la base de données
- Insérer de nouveau tuples
- Modifier des tuples
- Supprimer des tuples

Format de données

Le résultat d'une requête est défini comme une classe ayant trois attribues.

- `data` est un dictionnaire de liste d'objets avec pour clef le nom des champs. : `Dictionary<string, List<object>>`
- `succes` est à false si la requête génère une erreur sql
- `error_message` le potentiel message d'erreur

Toutes les requêtes sont envoyés à la base de donnée à l'aide de `DataBase.execute_query()`

L'avantage de ce format est qu'on peut traiter n'importe quelle requête tout et qu'il est très adapté pour une logique de table. En effet si on fait `SELECT * FROM Personne` il nous suffit de faire `qr["nom"][4]` pour récupérer le quatrième nom.

Exemple de l'insertion

Prenons l'exemple de la création de nouveaux tuples pour comprendre nos choix techniques :

Lorsque l'utilisateur clique sur le Menu « insérer de nouveaux tuples » on réalise une requête vers la table :

```
SELECT table_name FROM information_schema.tables WHERE table_
```

Cette dernière nous renvoie une liste de string avec le nom de chacune des tables. Notre Interface génère alors un menu partir de cette dernière. Si l'utilisateur clique sur une table pour la modifier on envoie une requête à la base pour récupérer le nom des différents champs et leur type. Grâce à cela on est capable de mettre en place une saisie utilisateur pour chacun d'entre eux.

Une fois les valeurs récupérées on génère la requête SQL adaptée pour l'insertion des valeurs choisis dans la table sélectionnée. Chaque type doit être géré de manière indépendante :

- int : rien à faire
- string : ajout des ''
- Date : Formatage pour la base de donnée

```
((DateTime)qr.data[key][0]).ToString("yyyy-MM-dd HH:mm:ss")
```

Autres fonctionnalités

Les opérations de modification de tuples et de suppression sont gérées de manière similaire. On assure les vérifications nécessaires au niveau des clefs étrangères. Si l'utilisateur tente d'attribuer une clef étrangère non définie alors on lui affiche celles existantes. Pour les supressions on a mit en place des DELETE ON CASCADE au niveau de la base de données.

Le Menu

Pour le menu nous avons crée une classe qui nous permet de naviguer d'un menu à l'autre et de générer automatiquement l'affichage à partir de liste de string obtenu via des requêtes sur la base. Cela parait anodin mais la fonctionnalité de retour en arrière était très compliquée à mettre en place.

Prise d'initiative

Nous avons mit en place une fonction qui permet à l'utilisateur de faire directement des requêtes sur la base en écrivant en console ses requêtes SQL.