Epsitec.DataLayer

Le projet Epsitec.DataLayer est la couche qui s’occupe de toute la gestion des entités. Cela implique entre autres la création des tables dans la base de données, le chargement des données des entités, la sauvegarde de ces données après modification et un certain nombre de services utilitaires.

# Architecture

Ce projet contient deux classes principales qui servent de point d’entrées :

* DataInfrastructure
* DataContext

La classe DataInfrastructure gère la connexion physique à la base de données et les services, c’est-à-dire la gestion de connexions et de verrous haut-niveau dans la base de données par exemple. Elle permet également de créer des instances de la classe DataContext.

La classe DataContext gère quant à elle tout ce qui a directement trait aux entités, comme leur chargement et leur sauvegarde.

Par ailleurs, l’implémentation de ces fonctionnalités n’est pas réalisée directement dans ces classes mais déléguées à d’autres classes qui s’occupent chacune de la gestion d’une fonctionnalité bien précise.

# DataInfrastructure

La classe DataInfrastructure a deux buts principaux :

* La gestion des services
* La gestion des instances de DataContext

Chaque service est géré par une classe qui lui est propre il s’agit de

* ConnectionManager qui gère les connexions haut-niveau à la base de données
* LockManager qui gère les verrous haut-niveau stockés dans la base de données
* InfoManager qui gère des paires de clés/valeurs stockées dans la base de données
* UidManager qui gère des générateurs de séquence produits par la base de données
* EntityModificationLog qui gère un journal des modifications des entités.
* EntityDeletionLog qui gère un journal des destructions des entités

Parmi ces services, deux méritent une description plus poussée, il s’agit de la gestion des connexions et des verrous haut-niveau. L’idée est que chaque instance du programme crée une connexion haut-niveau à la base de données en lui donnant un identificateur qui lui est propre. Chaque instance du programme est responsable de signaler à intervalle régulier aux autres instances qu’elle est toujours en vie. Si une instance n’a pas donné signe de vie depuis un temps donné, elle est considérée comme morte et donc automatiquement déconnectée par une autre instance.

En plus de cela, chaque programme peut créer des verrous, qui sont simplement une association entre un nom de verrou et un compteur qui compte combien de fois il a été pris par l’instance du programme qui le possède actuellement. Un verrou est donc réentrant et ne peut être possédé que par une seule instance du programme à la fois. Lorsqu’une connexion est terminée (normalement par sa propre instance, ou de manière forcée par une autre instance), tous les verrous associés sont automatiquement relâchés.

La création et la destruction des instances de DataContext sont réalisées au moyen d’appels à une instance de la classe DataContextPool.

La connexion physique à la base de données est gérée par une instance de la classe DbInfrastructure qu’on doit passer au constructeur de DataContext. Toutes les requêtes SQL et les transactions se feront par l’intermédiaire de cette connexion. Cette instance de DbInfrastructure ne devrait pas être partagée entre plusieurs DataInfrastructure.

Cette classe contient aussi une instance de la classe EntityEngine dont le but est de fournir toutes les informations relatives au schéma des entités et à leur type. Comme la classe EntityEngine est thread-safe (elle ne contient que des objets en lecture seule qui ne sont jamais modifiés), il est possible et même recommandé de partager une seule instance d’EntityEntine entre toutes les instances de DataInfrastructure.

# EntityEngine

La classe EntityEngine contient plusieures classes qui ont chacune une responsabilité particulière.

* EntityTypeEngine est chargée de la gestion des métadonnées des entités. Quel est leur type, quelles sont leurs propriétés, etc.
* EntitySchemaBuilder est chargée de la création des instances de DbTable, DbColumn et DbTypeDef qui vont décrire les tables utilisées pour stocker les entités dans la base de données.
* EntitySchemaEngine est chargée de maintenir un cache de ces instances de TbTable et DbColumn pour pouvoir rapidement retrouver les instances de DbTable et de DbColumn qui sont concernées par une entité donnée.

# DataContext

La classe DataContext est le point d’entrée pour toute la gestion des entités. Comme d’habitude, elle n’implémente pas grand-chose elle-même, mais délègue la fonctionnalité à des classes spécialisées.

* EntityCache qui gère le cache en mémoire des entités déjà chargées par le DataContext, et contient des informations additionnelles au sujet de ces entités.
* DataLoader qui gère le chargement des entités depuis la base de données
* DataSaver qui gère la sauvegarde des modifications vers la base de données

Chaque instance de DataContext est indépendante des autres, et toutes les modifications qui y sont faites restent locales, tant qu’on ne fait pas un appel à la méthode SaveChanges() qui sauve les changements dans la base de données. A ce moment-là, si le DataContext est synchronisé avec d’autres, les modifications sont propagées vers ces autres instances de DataContext.

On peut configurer le DataContext lors de sa construction pour activer certaines fonctionnalités, comme le rafraichissement des entités, la virtualisation des entités nulles ou le mode lecture seule. Un DataContext en mode lecture seule ne permet pas la modification d’entités, mais est thread-safe et peut donc être partagés par plusieurs threads en même temps.

# Request

La classe Request sert à définir des requêtes à exécuter sur la base de données pour récupérer des entités. Elle contient un exemple, c’est-à-dire un graphe d’entités qui détermine des critères qui vont filtrer les entités du résultat. Ces critères sont dérivés des valeurs des propriétés des entités.

En plus de cela, on peut lui spécifier des conditions supplémentaires et des critères de tris, qui vont influencer sur les éléments et leur ordre dans le résultat.

L’exemple et les conditions déterminent un ensemble d’entités dans la base de données. Par défaut, c’est les entités qui correspondent à la racine du graphe de l’exemple qui sont retournées, mais il est possible de retourner d’autres entités en utilisant la propriété RequestedEntity.

# Chargement des données

Le chargement des données des entités depuis la base de données se fait toujours en deux phases :

* On utilise une instance de LoaderQueryGenerator pour générer la requête SQL et l’exécuter. Les données reçues en réponse sont mises dans des instances de la classe EntityData.
* On utilise une instance d’EntitySerializationManager pour instancier les entités et leur assigner les données.

La génération de la requête SQL est particulièrement complexe, car il faut traduire la requête (une instance de la classe Request) en requête SQL, ce qui n’est pas trivial. En particulier, la gestion des jointures entre les tables est délicate.

La seule subtilité de la déserialisation des données est que potentiellement toutes les données d’une entité ne sont pas chargées en mémoire. Dans ce cas, on remplace ces données par un proxy qui va aller charger la donnée à la demande, de manière paresseuse.

# Sauvegarde des données

La sauvegarde des données s’effectue en plusieurs phases distinctes :

* Générations de jobs de sauvegarde, qui décrivent les modifications (insertions, mises-à-jour et destructions d’entités) à faire dans la base de données.
* Exécution de ces jobs de sauvegarde.
* Traitement sur les entités en mémoire, pour supprimer des références à des entités détruites et indiquer que les valeurs actuelles sont leurs nouvelles valeurs originales.
* Génération de jobs de propagation des modifications vers les autres DataContext.
* Exécution de ces jobs de propagation.

# RequestView

La classe AbstractRequestView sert, comme son nom l’indique, à offrir une vue par-dessus une requête. Elle permet basique ment de compter les éléments qui correspondent à cette requête et ensuite d’y accéder par tranche. Il y a deux implémentations concrètes de cette classe :

* DependentRequestView qui utilise les mécanismes standards du DataContext pour accéder aux données. Il n’y a donc absolument aucune différence entre exécuter une Request directement avec le DataContext ou à travers le DependentRequestView, si ce n’est la facilité de le faire.
* IndependentRequestView qui utilise une transaction dédiée pour effectuer toutes ces requêtes. Par conséquent, les données retournées sont totalement indépendantes des modifications qui pourraient être faites dans la base de données.

# Subtilités

Il y a un certain nombre de subtilités dont il faut tenir compte dans cette couche

* La gestion des entités vides. Le DataContext permet de dire qu’une entité qui n’a encore jamais été persistée est vide. Dans ce cas, elle ne sera pas sauvegardée dans la base de données. Le fait qu’une entité vide peut devenir non vide a des implications sur les autres entités qui ne sont techniquement pas modifiées mais dont il faut resauver certaines propriétés dans la base de données. C’est le but du champ fieldsToResave
* La gestion des entités qui ont étés crées par le EntityNullReferenceVirtualizer. La gestion de ces entités se fait principalement dans la classe EntityNullReferenceVirtualizer, mais il y a aussi quelques traitements spécifiques dans cette couche, comme par exemple lors de la sauvegarde des entités.
* Les données de toutes les entités du DataContext peuvent être rechargées. Pour des questions d’optimisations, on stocke dans EntityCache des informations sur la version des entités qui a été chargée (EntityModificationLogId). On utilise aussi les entrées dans l’EntityDeletionLog. Cela permet de ne recharger que les données des entités qui ont été effectivement modifiées.