Ymox

Zeste de savoir

APEX : développement dans Salesforce

15 octobre 2022

Table des matières

Introduction

Depuis plus de deux ans maintenant, je suis appelé a faire du développement sur la plateforme CRM Salesforce. Pour ce faire, ils proposent un langage de programmation qui, pour en citer que je connais, ressemble à du Java. Je vous propose donc un tour d'horizon.

1. Un langage Java-like

Je disais donc que la syntaxe d'APEX ressemble à du Java. Jugez-en par vous-même avec l'extrait ci-dessous.

```
public class ExampleClass {
        public static void sayHello(String name) {
                System.debug('Hello ' + name);
        }

        public static void sayHello(String[] names) {
                System.debug('Hello ' + names.join(', '));
        }
}
```

Il y a cependant des différences notables.

1. Les chaînes de caractères sont encadrées par des guillemets simples `'`.
2. Pas de `System.out`, mais `System.debug(Object msg)`, qui écrit dans le journal uniquement.
   La méthode possède une surcharge prenant deux paramètres : `System.debug(System.LogggingLevel logLevel, Object msg)`, ce qui ne permet donc de changer que le niveau de journalisation.
3. La notion d'imports n'existe pas, il n'y a qu'un seul *namespace*.
4. Les listes typées et les tableaux de primitifs sont interchangeables.
   En effet, que vous utilisiez `String[]` comme ci-dessus ou `List<String>`, les résultats et comportements seront les mêmes.
5. Les types primitifs sont un peu différents :
   - Blob (plus ou moins un équivalent du `byte` en Java, utilisé notamment comme corps de requêtes réseau externe ainsi que pour gérer certains cas de cryptographie) ;
   - Boolean
   - Date
   - Datetime
   - Double (et pas de Float)
   - ID (pour la gestion spécifique des données du CRM en APEX, spécialisation de String)
   - Integer
   - Long
   - Object
   - String
   - Time

   A noter que les types sont à la fois objets et primitifs, et que la casse de leurs noms n'a pas d'importance. Il est cependant recommandé d'utiliser les versions avec majuscule.
6. Le jeu de classes est restreint par rapport à Java

Ces informations se retrouvent dans la documentation officielle , qui évidemment permettra aux plus curieux de creuser les diverses questions plus techniques.

Ça, c'était pour les similitudes et différences les plus simples.

> Si en Java il est possible de faire fonctionner votre code partout là où il y a une JVM, et que vous pouvez convertir en bytecode sur une machine équipée, **l'APEX n'est "compilé" que sur les serveurs de Salesforce**. Vous n'aurez donc jamais accès qu'aux sources et métadonnées, jamais au code "compilé".
> Cela implique donc que même pour tester quelques lignes de code, il vous faudra être connecté à une instance de Salesforce. Sans cela, impossible de tester du code.

Si après la lecture de ce billet vous souhaitez tester un peu le langage, il semble qu'il soit possible d'avoir accès à des instances de test.

2. Manipuler des objets de Salesforce en APEX : le SOQL

Salesforce est avant tout un CRM, donc le code qu'on écrit pour la plateforme est appelé à utiliser des données. Une des différences importantes avec Java sur ce point est que **la syntaxe de base d'APEX comporte un SQL** évidemment adapté à la structure des données, appelé **SOQL**. Dans la mesure ou les données sont organisées sous forme d'objets, le SOQL est donc un SQL où l'on a ajouté un O pour Objet.

En règle générale, on va prendre une syntaxe SQL pour une requête `SELECT`, remplacer les noms de tables par des noms d'objet et les noms de colonnes par des noms de propriétés, ce qui donne quelque chose comme suit.

```
1 Date min_date = new Date();
2 List<Contact> contacts = [SELECT Id, FirstName, LastName FROM
      Contact WHERE Birthday < :min_date];
```

> Il est obligatoire de lister explicitement les propriétés à récupérer, le "joker" `*` de SQL n'existe pas

La syntaxe pour mettre une requête dans du code est simplement de la mettre entre crochets.

Il y a évidemment quelques subtilités, comme une liste restreinte d'opérateurs de comparaison (une fois n'est pas coutume, mieux vaut se référer à la documentation officielle à ce propos) et une syntaxe particulière pour effectuer les jointures.

2.1. Les jointures, parlons-en

Les objets, c'est bien, mais quand ils sont reliés entre eux, c'est plus pratique. Il est évidemment possible de récupérer les objets liés à d'autres en SOQL, mais là, pas question d'utiliser `INNER JOIN`, `LEFT JOIN` et encore moins `RIGHT JOIN`. En fait, la syntaxe va dépendre de la cardinalité de la relation entre les deux objets à joindre.

Les relations entre objets sont toujours bi-directionelles dans Salesforce

2.1.1. Jointure sur relation *ToOne

En cas de relation *ToOne d'un objet vers un autre, on va déjà commencer par ajouter dans la liste des propriétés à récupérer le nom de celle qui représente la relation. Prenons la relation entre Adress et Contact qui est justement une ManyToOne. Voici ce que cela donnerait de prime abord.

```
1 List<Address> addresses = [SELECT City, Country, Contact FROM
      Address];
```

Le problème avec cela, c'est qu'on doit lister explicitement les propriétés à récupérer. Or, si Contact est bien une propriété de l'objet Address, on ne liste pas les propriétés de Contact. Comment fait-on en SQL pour spécifier qu'on souhaite un champ d'une autre table, notamment en cas de champs ayant le même nom dans deux tables jointes ? On préfixe avec le nom de la table. Ici, on va donc mettre le nom de l'objet lié suivi du nom de la propriété qu'on souhaite récupérer, de la même manière qu'on accéderait à la propriété dans du code habituel.

```
1 List<Address> addresses = [SELECT City, Country,
      Contact.FirstName, Contact.LastName FROM Address];
```

A noter :

- on ne spécifie pas qu'on va récupérer deux types d'objets, la clause `FROM` ne comporte que l'objet principal ;
- cette syntaxe peut aussi s'utiliser dans les contraintes, voir ci-dessous.

```
1 List<Address> addresses = [SELECT City, Country,
      Contact.FirstName, Contact.LastName FROM Address WHERE
      Contact.Birthday < :min_date];
```

Et si je veux joindre un autre objet en relation *ToOne avec Contact, cette fois ?

De la même manière qu'on a utilisé `Contact.FirstName`, on va continuer à chaîner !

```
1 List<Address> addresses = [SELECT City, Country,
      Contact.FirstName, Contact.LastName, Contact.Department.Name
      FROM Address WHERE Contact.Birthday < :min_date];
```

> *i* La documentation explique qu'il est possible de chaîner ainsi 4 niveaux. Je n'ai jamais rencontré le cas ni eu la possibilité de tester si les niveaux incluent la propriété (3 opérateurs d'accès `.`) ou s'il s'agit des objets uniquement, donc 4 opérateurs d'accès `.`

2.1.2. Jointure sur relation *ToMany

Evidemment, il n'y a pas que les relations *ToOne qui sont disponibles. Si on voulait prendre les requêtes ci-dessus dans l'autre sens, comment cela se présenterait-il ?
La notation `SousObjet.SousSousObjet.Propriété` fonctionnait bien dans les relations *ToOne parce qu'on savait qu'on n'aurait justement qu'un SousObjet par objet principal. Là, on va par définition en avoir plusieurs. Comment gérer cela ? On va simplement utiliser un mécanisme qui existe aussi en SQL : les sous-requêtes.

Ainsi, si l'on "changeait le sens" de la première requête correcte pour les relations *ToOne, on arriverait à cela.

```
1 List<Contact> contacts = [SELECT FirstName, LastName, (SELECT
      City, Country FROM Addresses) FROM Contact WHERE Birthday <
      :min_date];
```

Tant qu'on est dans la partie `SELECT`, on utilise des propriétés, *aussi pour les clauses `FROM` des sous-requêtes*. Il n'est donc pas possible de descendre "plus bas" qu'un niveau avec cette syntaxe, parce que la clause `FROM` d'une seconde sous-requête dans la première ne pourrait pas faire référence à une propriété de l'objet principal Contact. Par contre, cela explique en partie la présence de `Addresses` dans la sous-requête : c'est le nom de la propriété qui permet d'aller dans le sens One(Contact)ToMany(Address), vu que les relations sont bi-directionnelles.

2.2. Requêtes préparées

Evidemment, dès qu'il y a un langage de requêtage, il faut un mécanisme pour y insérer des valeurs. J'ai volontairement glissé un exemple dans l'extrait de code en début de section : `:min_date` à la seconde ligne est un marqueur dans la requête. La syntaxe est simple : un deux-points `:` suivi de plusieurs caractères ASCII *correspondant à une variable existante*, ici celle déclarée à la ligne 1.
A noter que les propriétés étant typées et que APEX est un langage fortement typé, ce genre de mécanisme impose que les valeurs utilisées dans des contraintes sur une propriété soient de

même type que ladite propriété. Si la variable `min_date` était de type Datetime, la requête n'aurait pas pu être exécutée.

Un des avantages de la syntaxe est qu'elle est utilisable avec l'opérateur `IN` quand la variable est de type `List<T>`, avec `T` compatible avec le type de la propriété utilisée dans la condition.

2.3. Les autres opérations

On vient de voir dans bien des sens comment sélectionner des données, il faut aussi pouvoir les mettre à jour et les supprimer. Pour cela, pas de requêtes `INSERT`, `UPDATE` ou `DELETE`, mais des mots-clés du langage APEX `insert`, `update` et `delete`, à faire suivre d'un objet ou d'une liste d'objets. Ces derniers doivent à ma connaissance être du même type, je ne crois pas qu'il soit possible de faire une liste d'objets générique et de demander à ce qu'ils soient tous mis à jour.

3. Des objets plus spécifiques

APEX est pensé pour traiter des données, qui sont représentées sous forme d'objets, dont je vais parler tout soudain. Commençons par les objets "de stockage".

3.1. Les objets de stockage (équivalents des POJOs)

Il y a beaucoup d'objets fournis dès le départ. Salesfore étant un CRM, on trouve évidemment de quoi gérer les clients (l'objet Contact utilisé précédemment), les produits, etc. Par contre, il n'est pas rare d'avoir des besoins plus spécifiques et de nécessiter un nouvel objet.

Dans un premier temps, il peut être bon de savoir que tous les objets de données héritent d'un type **`sObject`** ("**S**torage **object**" ?). Celui-ci possède quelques champs qui, de fait, se retrouveront dans tous les objets de stockage. En voici quelques uns:

- `Id`, dont la fonction est explicite. A savoir que les IDs dans Salesforce sont des chaînes de 18 caractères de long, les trois derniers servant de somme de contrôle;
- `Name`, qui sert plus ou moins d'identifiant unique visuel pour un objet. C'est un des champs qui est automatiquement indexé pour la recherche globale sur une instance Salesforce;
- `LastModifiedDate`, dont le nom est explicite;
- `LastModifiedBy`, qui retourne l'Id de l'utilisateur ayant effectué la dernière modification. A noter que cette propriété peut être utilisée pour du chaînage afin notamment de récupérer le nom de l'utilisateur: `LastModifiedBy.Name`;
- `CreatedDate`, pour qui le nom est aussi explicite;
- `CreatedBy`, équivalent de `LastModifiedBy` pour la date de création.

3.1.1. Création de nouveaux objets

Contre toute attente, quand on a besoin de nouveaux objets, ceux-ci ne peuvent pas être créés avec du code APEX en définissant la classe, comme il serait de mise de le faire avec des ORMs courants. Deux solutions:

— ajouter la définition de l'objet (correspondant plus ou moins aux mappings des ORMs) dans un fichier de métadonnées,
— ou passer par l'interface de génération fournie dans Salesforce.

Parmi les informations nécessaires à la création, Salesforce demande de fournir deux versions du nom:

1. une "humanisée", où l'on peut mettre quelque chose de lisible couramment (et donc des accents et des espaces) qui sera utilisée dans les interfaces de gestion ;
2. et une version "machine" contenant uniquement des caractères ASCII visibles.

Mais la manière d'y accéder dans les programmes doit tenir compte d'un petit détail: le nom machine choisi pour l'objet doit être suffixé par `__c`. Ainsi, un nouvel objet pour lequel le nom humanisé serait "Cross Department Identifier", le nom machine saisi dans l'interface graphique serait `CrossDepartmentIdentifier` mais dans le code APEX, on devrait s'y référer avec `CrossDepartmentIdentifier__c`.

?

Mais alors, dans les exemples que tu as donné pour les requêtes, pourquoi n'as-tu pas utilisé `Contact__c`, `Address__c` et `Department__c`?

Parce que cette logique ne s'applique qu'aux objets personnalisés, ou "**C**ustom" en anglais. Or, les objets Address, Contact et Department sont des objets fournis par Salesforce. Le suffixe qui indique que ce sont des objets personnalisés n'est pas de mise.

i

La logique du suffixe s'applique aussi aux champs personnalisés.

Ainsi, quasiment tous les champs d'un objet personnalisé devront prendre le suffixe, hormis les champs de `sObject`. Dans les objets fournis par Salesforce, seuls les champs personnalisés de l'objet auront ce suffixe. ce sont là quelques subtilités qui peuvent surprendre au début.

3.2. Observer le cycle de vie des objets avec les triggers

Il est normal que les données d'un CRM vivent, et donc il peut être nécessaire d'observer les création, mises à jour et suppression des objets pour automatiser certains traitements. Pour cela, le langage propose un type d'objet spécialisé: les **triggers**. Prenons tout de suite un exemple de déclaration.

```
1 trigger ContactChange on Contact (after create, before update,
      after delete, after undelete) {
2     // …
3 }
```

Listing 1 – Exemple de déclaration d'un trigger

— On utilise donc le mot-clé `trigger` au lieu de `class`.
— On spécifie quel objet est concerné avec `on` plus le nom de l'objet.
— On définit quels "moments" nous intéressent avec une liste entre parenthèses.

Il y a 7 "moments" disponibles, composés d'un "instant" et d'une "action":

— avant
  — création
  — mise à jour
  — suppression
— après
  — création
  — mise à jour
  — suppression
  — "désuppression" (correspondant à une sortie de corbeille).

Ces moments se construisent avec les mots-clés `before` et `after` pour l'instant, puis viennent les actions `create`, `update`, `delete` et `undelete`. Le mot-clé `before` rend possible de modifier les valeurs d'un objet avant qu'elles soient enregistrées, et l'on pourrait interrompre l'action en cas de souci. "L'instant" `after`, lui, permet de récupérer les valeurs de champs dynamiques, ces valeurs n'étant calculées que lorsque l'action a réellement été effectuée.

4. Vraiment beaucoup d'enregistrements ? Voici les BigObjects

Il ne faut pas oublier que toute information à enregistrer prend de l'espace de stockage, et c'est un des nerfs de la guerre pour Salesforce. Quand l'espace vient à manquer, on peut évidemment en acheter plus, mais les prix deviennent vite conséquents. Du coup, ce type d'objets a deux avantages notables.
Déjà non, le nom de veut pas dire que l'on peut mettre plus de propriétés dans ces objets: l'idée est d'avoir des performances optimales bien au-delà d'une dizaine de millions d'enregistrements!
Ensuite, l'espace de stockage pour des enregistrements de ces objets est compté à part et est bien plus intéressant niveau budget que ce qui serait nécessaire si tout était fait en objets "conventionnels".
Cependant, on se doute bien qu'il y a un revers à la médaille: les BigObjects ont un comportement différent.

Réfléchissez ***VRAIMENT*** à l'utilité et l'utilisation d'un BigObject avant d'en créer un. Rendez-vous ce service : considérez les contraintes et agissez en conséquence avant qu'il n'y en ait de plus ennuyeuses.

4.1. Pas de champ ID, mais un groupe de champs qui forment un index unique

Une des différences les plus importantes est que ces objets n'ont pas un champ ID propre comme ceux présentés plus haut, du moins pas accessible. En revanche, ce qui permet de les identifier est un groupe de champs (défini à la création), prendre cela comme une clé primaire composite.

4.1.1. On ne peut en choisir que 5 maximum

On pourrait évidemment ne mettre qu'un champ dans le groupe, mais on verra plus loin que ce n'est pas pratique.

4.1.2. Le groupe ne doit pas faire plus de 100 caractères

Techniquement, il semble que les valeurs du groupe soient concaténées en une seule chaîne et celle-ci serait utilisé comme identifiant unique interne. Et voilà du coup que cette valeur fantôme possède une limite : il faut donc faire attention aux champs choisis pour composer cet index.

i Taille de certains types

Pour les champs textuels, il faut compter la longueur maximale ; pour les types non textuels, le nombre de caractères pris en compte dans la taille de l'index utilisera la définition du champ. Pour les dates et heures, c'est la représentation ISO 8601 qui sert, mais avec la précision des millisecondes et une indication du fuseau horaire si nécessaire.

Type	Définition	Nombre de caractères dans l'index
Nombre	18 chiffres, dont 2 décimales	18 caractères
Heure	ISO 8601 HH:mm:SS,sss	12 caractères
Date	ISO 8601 AAAA-MM-JJZZZZZ	15 caractères
Date et heure	ISO 8601 AAAA-MM-JJTHH:mm:SS,sssZZZZZ	28 caractères – *plus d'un quart de l'index !*
ID	texte	18 caractères

4.1.3. Tous les types de champs ne peuvent pas être dans le groupe d'index

Vu la limite des 100 caractères, on comprend aisément qu'on ne puisse pas utiliser de type texte long, qui comporte jusqu'à 131 072 signes. On ne peut pas non-plus utiliser de références—mais on peut en revanche mettre des IDs sous forme de texte à 18 caractères.

4.2. On ne peut rechercher que sur les champs de l'index…

Oui, voilà, la performance pour un grand nombre d'enregistrements a ce coût: un champ qui n'est pas indexé ne pourra pas servir dans une requête, là où quasiment tous les champs des objets usuels peuvent être source de contraintes. Vous devez faire des recherches sur plus de champs que les 5 maximum? Une possibilité serait d'avoir des tuples de BigObjects avec des références les uns aux autres, comme si vous "éclatiez" les informations.

4.3. … et dans l'ordre de déclaration s'il vous plaît !

Imaginons que vous avez un BigObject qui enregistre les ordres d'envoi de colis pour les clients, BigObject pour lequel vous avez choisi des champs pour l'index:

1. la date de création;
2. l'ID du produit;
3. l'ID du client.

Maintenant, vous voulez trouver quand chaque produit commandé a été envoyé à un client en particulier. En l'état, c'est impossible: **le système ne permet pas de rechercher par un seul des champs d'index sans que les précédents n'aient été eux aussi mis dans le critère**. On ne peut donc même pas chercher quand un certain produit a été envoyé aux clients qui l'ont commandé sans avoir la date…
Du coup, **la construction de l'index doit être réfléchie pour que les recherches ensuite ne soient pas problématiques**. *Exit* donc une logique de propreté, il faut prendre en compte une vraie logique d'utilisation, et encore, elle devra s'adapter. Dans le cas qui nous occupe, est-ce qu'on veut le plus souvent savoir:

— quels produits on a envoyé quand à un client (et donc l'ID du client serait en première position dans l'index)?
— à qui on a envoyé un produit en particulier et quand (l'ID du produit prime)?
— quand a-t'on envoyé quelque chose à qui (la date est plus importante)?

Et ce questionnement est évidemment à refaire pour le second élément de l'index, ainsi que les suivants.

4.4. Une fois créés, on n'y touche plus !

Si les objets les plus courants peuvent être modifiés alors qu'ils sont utilisés, **une fois un BigObject défini et mis à la disposition du système, on ne peut plus en modifier sa structure de quelque manière que ce soit**. Vous n'aviez pas réfléchi correctement à l'ordre des composants de l'index? Vous avez oublié une propriété, ou ne l'avez pas typée correctement? *Vous recréez l'objet de A à Z.* Si par malheur vous aviez en plus déjà enregistré des informations avec la mauvaise structure, vous devez tout migrer vous-même dans la nouvelle. Evidemment, le nouvel objet ne peut pas avoir le même nom machine que le précédent, d'autant plus si vous devez migrer des informations, donc le code que vous auriez fait doit être adapté…

4.5. Les interfaces automatiques présentes pour les objets standard ne sont pas disponibles

En reprenant l'exemple précédent des envois de produits, si vous devez afficher les informations de ces BigObjects, c'est possible, mais **c'est une interface à coder**. Vous voulez quelque chose d'un peu convivial avec des filtres? Codez-la. Et je ne parle pas des informations qui doivent être accessibles a certains rôles utilisateur, mais pas à d'autres.
Bref, là où il y a beaucoup d'outils no-code, voire des vues par défaut un tant soit peu personnalisables, pour l'interface des objets standards, il n'y en a pas pour les BigObjects.

4.6. La gestion de la persistance est différente

Pas moyen de faire `insert monBigObject`, les mots-clés utilisés par les objets standard ne sont pas adaptés. Au lieu de cela, il faut utiliser l'objet Database et sa méthode `insertImmediate()`.

Cela a évidemment des conséquences…

4.7. Impossible de leur adjoindre des triggers

Hé oui, les BigObjects ne déclenchent pas d'événements qui puissent être écoutés, du fait de la manière différente de les persister. Attention cependant : cela ne veut pas dire qu'on ne peut pas créer de BigObject dans un trigger d'un objet courant.

4.8. Les insertions font des mises à jour quand l'index existe déjà

Plutôt que de jeter une erreur quand un index est déjà présent, le moteur considère que c'est une mise à jour. Toujours avec notre exemple d'envoi de produit, imaginons que pour une raison ou une autre, vous avez dû envoyer deux fois le produit au même client le même jour. Ben vous n'aurez qu'un enregistrement si vous n'aviez mis que la date sans heure dans l'index.

4.9. Le suffixe du nom d'objet est différent

Cela, c'est un peu du détail, mais c'est une des différences : là où le nom machine des objets personnalisés se terminait par `__c`, pour les BigObjects, on aura `__b`, tout simplement.
Attention : cela n'a pas d'impact sur les noms de champs dans les BigObjects, qui conservent le suffixe `__c`.

> *i* Je l'avais dit, prenez le temps de bien considérer ces contraintes avant de créer et d'utiliser les BigObjects.

5. Des limites inamovibles

On arrive maintenant à l'un des points auxquels il faut particulièrement faire attention quand on travaille en APEX : les différentes limites d'exécutions—et il y en a beaucoup. Je ne vais pas faire une liste exhaustive, la documentation s'en chargeant mieux que moi , mais je vais en expliciter quelques unes. Et avant cela, il me faut encore préciser quelques points.

— **Les limites ne sont pas modifiables/négociables.**
Si les langages de programmation comme PHP permettent de modifier par exemple la mémoire allouée ou le temps maximal d'exécution, il n'est à ma connaissance pas possible de personnaliser (ni de *faire personnaliser*) quoi que ce soit à ce niveau pour une instance déterminée.

— **Les limites sont appliquées *par transaction*.**
L'APEX est aussi pensé comme un langage de bases de données : une exécution de code est donc à mettre en parallèle avec celle d'une procédure stockée : les deux sont dans une transaction au niveau base de données.

— **Les limites peuvent changer selon le contexte d'exécution synchrone ou asynchrone.**
Ici, les exécutions synchrones seraient celles gérées par les triggers vu précédemment, et les exécutions asynchrones celles de l'équivalent des tâches CRON de Salesforce. Je ne vais volontairement pas mentionner de valeur dans ce billet, l'idée étant plus d'attirer l'attention sur l'existence des limites et de mentionner les solutions qui ont été mises en place dans le cadre de mon travail, mais du coup il est intéressant de garder à l'esprit que

l'un ou l'autre contexte propose des limites plus larges – habituellement les traitements asynchrones.

5.1. Mémoire d'exécution maximale

Ce qui est à prendre en compte quand on a de gros volumes de données à traiter (recalcul de TVA, par exemple). Si on touche à de gros objets ou beaucoup d'objets de taille moins importante, le volume de données peut vite poser problème. Du coup, autant limiter les requêtes SOQL aux champs réellement nécessaires.

5.2. Temps d'exécution maximal du code

Toujours dans le cas de gros volume, mais aussi de traitements complexes. Parfois, il vaut mieux prendre des lots plus petits et traiter plus souvent que prendre des lots aussi gros que possible, mais ne pas pouvoir terminer le traitement. Pour de très gros volumes et des traitements complexes qui ne peuvent pas être facilement séparés, il existe des solutions qui vont permettre de scinder en autant de transactions que nécessaire le traitement total.
Par contre, **le temps d'exécution des requêtes SOQL n'entre pas ici en ligne de compte**, c'est vraiment l'exécution de la logique du code qui est chronométrée. La requête prend cinq à six secondes pour agréger des résultats, mais le code ne prendra qu'une ou deux secondes à les traiter ? Il sera considéré que l'exécution aura duré une ou deux secondes.

5.3. Nombre maximal de requêtes SOQL

Hé oui, le nombre de requêtes effectuées est limité dans une transaction. Ce qui implique que les requêtes doivent être optimales, et utiliser le plus possible les jointures.
Seulement, si la manière de faire avec les relations *ToOne fait que c'est compté comme une seule requête, **la manière de faire avec les relations *ToMany compte comme deux requêtes**, trois si l'on imagine prendre les contacts, leurs adresses et leurs commandes, etc. Si on reprend les exemples précédents, sélectionner un contact puis dans une autre requête ses adresses revient au même que d'avoir les sous-requêtes pour ce qui est du décompte. Le seul avantage que procure la seconde méthode est lorsqu'on va récupérer plusieurs contacts : les données liées sont forcément déjà regroupées par contact.

5.4. Nombre maximal d'objets traités, toutes requêtes SOQL confondues

Et là c'est un petit piège quand on utilise les jointures, surtout avec les sous-requêtes. Si l'on va récupérer une série de contacts avec leurs département, *on double le nombre d'objets récupérés.* Si on utilise une sous-requête, *on ne sait pas nécessairement combien d'objets on va récupérer en plus.* Il faut donc bien calibrer les requêtes. Pour les sous-requêtes, il est tout à fait possible de leur ajouter une clause `WHERE`. En revanche, il n'est pas possible d'y mettre une clause `LIMIT`. Cela va forcément influencer l'architecture du code, sinon la portée de ce qu'on est en train de réaliser. Il n'a pas été rare d'avoir dû chaîner des exécutions dans le code que j'ai réalisé ces deux dernières années.
Pour l'anecdote : dans le cas de requêtes avec des regroupements, il ne faut pas non plus que le nombre d'objets à traiter dépasse la limite *avant regroupement.*

5.5. Nombre maximal d'objets créés, mis à jour et supprimés.

Encore un point auquel faire attention. Si on peut sans autre récupérer un grand nombre d'enregistrements, avec l'idée de tous les modifier, on ne pourra pas nécessairement persister toutes les modifications en une fois, il faut scinder en lots plus petits…

5.6. Nombre maximal d'opérations de création, mise à jour et suppression

… et vraiment ne pas en avoir trop, parce que là aussi il y a une limite. Heureusement, si on ne fait pas ces opérations de manière unitaire dans une boucle, mais qu'on regroupe par opération, ce n'est pas un souci. Ainsi, beaucoup de scripts sur lesquels j'ai travaillé ont des listes d'objets à créer, à supprimer et à mettre à jour, et les opérations effectives se font en fin de code, après les boucles de traitement, ces dernières ajoutant les objets dans les bonnes listes au fur et à mesure.

5.7. Nombre d'appels externes

Tout ce qui a été développé comprend évidemment un journal d'exécution. Or, la solution proposée par Salesforce ne permet pas aisément de s'y retrouver, on a donc décidé d'envoyer ces données ailleurs, et d'utiliser pour cela l'API proposées par un agrégateur de journaux de logs, API accessible par HTTP. Mais la manière de gérer ces appels externes est particulière, et fait l'objet d'une limitation en nombre. Seulement, pour pouvoir aider à déboguer (ce qui reste un des buts premiers des journaux de logs), on a préféré conserver dans certains cas l'envoi dans des boucles afin d'avoir les informations au plus proche du problème que ne rien avoir du tout. De plus, certains événements ne sont pas présents uniquement dans les journaux de logs, mais envoyés à des services de surveillance, et là aussi l'envoi doit être fait au plus proche de la réalité. Malheureusement, le volume de données augmentant, nous avons dû augmenter la taille des lots traités dans les limites des autres contraintes, mais il nous a fallu limiter les envois. Certains comportements qui avaient clairement été définis comme des erreurs à signaler ont été bâillonnés afin d'éviter d'atteindre cette limite.

5.8. Couverture minimale de code par les tests unitaires

Il est possible de rédiger des tests unitaires en APEX, et c'est même obligatoire pour le déploiement sur les instances de production. Ainsi, si un test échoue, le code n'est pas déployé. Si la couverture de code n'est pas suffisante, le code n'est pas déployé. Si au départ, et malgré la situation dans laquelle le projet a commencé, la couverture de code était très bonne, il faut s'en méfier quand le temps passe. Les *features* s'enchaînent avec leurs délais, il faut mettre en production le plus rapidement possible donc on met de côté les tests unitaires, et vient un moment où on ne peut plus mettre les nouveautés en production parce que la couverture de code a trop diminué.
Evidemment, la solution de mettre du code "de test" qui ne fait que couvrir le code "final" est tentante, mais c'est tout aussi piégeux, parce que justement le comportement n'est pas vérifié. C'est là que le *Product Owner* a toute son importance à mon avis : dans la mesure où il va tester les cas auxquels il pense, il va forcément prévoir un jeu de données qu'il est facile de reproduire dans les tests.

Conclusion

Voilà, vous avez désormais quelques notions plus ou moins avancées du développement qu'il est possible de faire dans le cadre d'un des CRMs parmi les plus utilisés au monde. Malgré les différentes restrictions qui peuvent sembler rébarbatives, j'ai été content de me mettre dedans et de voir que le peu de Java que j'ai fait soit revenu assez vite, le plus difficile pour moi ayant été de changer parfois de paradigme de programmation pour justement éviter les problèmes avec les limites.
En effet, certaines tâches demandaient de charger beaucoup d'objets différents, et les liaisons ne permettaient pas d'avoir suffisamment la main sur le nombre retourné, il a donc fallu effectuer des requêtes séparées pour avoir la finesse recherchée dans le volume, et reconstruire certaines relations avec des tables de correspondance. C'est dans ce genre de cas qu'on peur apprécier la souplesse que procure un ORM pour un langage qui permet des réglages machine plus souples.

Liste des abréviations

POJOs Plain Old Java Objects. 1, 6