

Bonnes pratiques pour les accès aux données

par Johann Blais

Date de publication : 06/02/08

Dernière mise à jour : 06/02/08

Cet article a deux objectifs :

- 1 Présenter quelques manières d'écrire un code générique pour les accès aux données (.NET 1.1 et 2.0)
- 2 Rappeler et expliquer les avantages des paramètres SQL dans le cadre des requêtes SQL (et des procédures stockées).

- I - Introduction
- II - Abstraction des classes en .NET 1.1
- III - Abstraction des classes en .NET 2.0
- IV - Utilisation de requêtes paramétrées
 - IV-A - Le problème
 - IV-B - La solution
- V - Conclusion
- VI - Remerciements

I - Introduction

Les bases de données sont aujourd'hui monnaie courante, et ce depuis un certain temps d'ailleurs. Avec chaque nouveau langage, arrive son lot de nouvelles fonctions, de nouvelles manières de travailler avec des bases.

Nous (développeurs, analystes, etc) avons en nous ce qui à mon avis est notre plus grande qualité et notre plus grand défaut, nous sommes paresseux. Avant de me jeter des cailloux, laissez moi développer un peu sur ce point. Vous vous êtes tous dit un jour ou l'autre, "j'aimerais bien faire ça, mais bon il y a sûrement déjà quelqu'un qui a eu la même idée, et qui, dans sa grande bonté, a décidé d'en faire profiter la communauté"; on peut d'ailleurs y voir l'origine du "Google is your friend". Disons les choses comme elles sont, dans ce cas, être fainéant c'est bien, c'est même très bien.

Mais, car il y a toujours un mais, cette qualité se retourne souvent contre nous. J'en prends pour exemple ce collègue que nous avons tous, celui qui fait partie des dinosaures, a connu l'informatique à ses débuts, épluchait les listings en LISP et en ADA des magazines de l'époque; je suis sûr que vous voyez de qui je parle. Et bien, ce collègue dont vous riez en cachette (ou ouvertement pour les plus téméraires), est un feignant dans le mauvais sens, il a gardé ses principes de développement anté-diluviens.

Le problème est que l'on a tendance à prendre nos petites habitudes, assez vite les bonnes et malheureusement très vite les mauvaises.

J'écris cet article avec pour objectif de vous donner quelques trucs, quelques voies à emprunter pour vous faciliter la vie ou plutôt le débogage. Car avoir les bonnes méthodes dès le début vous permet de gagner en rapidité, et surtout en efficacité.

Je commencerai par vous parler d'abstraction des méthodes d'accès, et je terminerai par les bonnes méthodes pour exécuter une requête en fonction de la situation.

II - Abstraction des classes en .NET 1.1

Nombre d'entre vous ont déjà développé des projets impliquant plusieurs sources de données, comme par exemple une base Oracle d'un côté, une base SqlServer de l'autre, et souvent plusieurs bases de chaque. Vous savez donc combien il peut être fastidieux de toujours avoir à ouvrir une connexion, créer une requête, l'exécuter, parcourir les données et... recommencer avec la base suivante. Car la plupart du temps, vous devez récupérer de nombreuses données en même temps et à plusieurs endroits.

Résultat, vous avez cinquante lignes de code qui servent à récupérer les données, et vous n'avez même pas encore effleuré l'aspect métier de la méthode. Cette approche peut fonctionner dans le cadre d'un projet "Papa/Maman", mais dans la vraie vie, vous en conviendrez, ce n'est pas ce qu'on peut qualifier de "professionnel".

Pour ceux d'entre vous qui s'impatiente, j'en viens à ce qui vous intéresse : le CODE. Je ne vais cependant pas vous submerger avec des lignes et des lignes à n'en plus finir. Mais intéressons nous à certaines structures disponibles dans le framework.

Les framework .NET (toutes les versions) possèdent dans le namespace *System.Data*, toute une collection d'interfaces destinées à nous faciliter la vie. Voici la liste des plus utiles :

- a IDbConnection
- b IDbCommand
- c IDbParameter
- d IDbTransaction
- e IDbDataAdapter...

La plupart d'entre vous auront déjà reconnu les types d'objet qu'ils manipulent quotidiennement mais dans leur forme concrète (*SqlConnection*, *OracleDataReader*, etc). Quel est alors l'intérêt pratique d'utiliser les interfaces plutôt que les implémentations. Précisons d'abord que même en utilisant les interfaces, il faut bien à un moment ou un autre leur associer une implémentation (SqlServer, Oracle, autre), mais nous reviendrons sur ce point plus tard.

L'utilisation des interfaces permet d'écrire un code qui ne dépend pas de la base de données sur laquelle il sera exécuté. Je suis sûr que certain(e)s d'entre vous ont déjà eu à changer de base de données en cours de projet. Si on n'est pas bien préparé, ça peut vite tourner à la dépression nerveuse. Passons de suite à un exemple concret.

Vous êtes un développeur prévoyant, vous avez déjà créé une classe qui vous renvoie une commande initialisée à partir d'une chaîne de caractères contenant le code SQL, et comme vous êtes encore plus prévoyant, vous passez aussi à votre méthode une enum qui permet même de sélectionner la source de données de destination. Vous avez donc une méthode qui a cette tête là :

```
public SqlCommand CreateCommand(string commandText, connectionType dataSource);
```

Là j'ai envie de vous dire "c'est pas mal", c'est déjà mieux que ce qu'écrivent 90% des développeurs.

Vous utilisez donc joyeusement cette méthode partout dans votre code, et vous avez raison. Le problème c'est que comme tout le monde, vous avez un chef qui a tendance à courir en agitant les bras dans tous les sens au moindre problème. Et là justement le problème c'est que SqlServer, on en a un peu assez, les clients veulent du Oracle. Qu'à cela ne tienne, vous migrez votre base, et vous attaquez maintenant à modifier votre code. Vous transformez évidemment votre précieuse méthode en :

```
public OracleCommand CreateCommand(string commandText, connectionType dataSource);
```

Et là, c'est le drame, il faut modifier le code à tous les endroits qui appellent cette méthode (heureusement il n'y en a que 1500), sans compter les `SqlDataReader`, `SqlParameter`, etc à modifier par la même occasion.

Ça fait mal. Revenons au moment de la création de la méthode *Flashback*. Vous savez que *OracleCommand* et *SqlCommand* implémentent l'interface *IDbCommand*. Au lieu d'utiliser un type spécifique, vous allez modifier votre PM (précieuse méthode) pour qu'elle renvoie un *IDbCommand* à la place. La signature de votre PM devient :

```
public IDbCommand CreateCommand(string commandText, connectionType dataSource);
```

Je précise aussi que vous ne modifiez rien d'autre dans la PM, simplement le type de retour dans la signature. Vous utilisez votre PM aux mêmes endroits et comme vous utilisez des *IDbCommand* dans la PM, vous continuez sur votre lancée, et utilisez des *IDbParameter* et des *IDataReader*, partout où il faut.

On revient au point critique, le changement de base de données. Au début même chose, migration de la base de données, etc. Et vous en arrivez aussi au moment de modifier votre code. Vous commencez aussi par votre PM. Le type de retour ne change pas, *OracleCommand* implémente aussi *IDbCommand*. Il suffit de remplacer dans le corps de la PM, les objets `Sql*` par des objets `Oracle*`.

Et là, miracle, c'est tout. Tout fonctionne exactement pareil. Pas de rechercher/remplacer sauvage, pas de prise de tête, changements réduits au minimum. Bref votre chef est content !

A ce moment, vous avez fait un constat encore plus accablant, le changement de base de données peut n'avoir pour conséquence que le changement de l'objet connexion. En effet, on peut créer une commande en utilisant la méthode *CreateCommand* de l'interface *IDbConnection*. En admettant que vous soyez familier avec le design pattern "factory", vous avez compris que le changement de base de données peut n'avoir comme un impact que l'ajout d'un case dans la méthode qui crée la connexion en fonction de la source de données.

Voici un exemple plus complet :

```
public IDbConnection CreateConnection(connectionType type)
{
    IDbConnection connection = null;
    switch(type)
    {
        case connectionType.MaBaseOracle :
            connection = new OracleConnection(maChaineDeConnexion);
        case connectionType.MaBaseSqlServer :
            connection = new SqlConnection(maChaineDeConnexion);
    }
    return connection;
}
```

Pour ceux qui ne veulent pas forcément gérer plusieurs sources de données, il suffit de simplement de supprimer le switch et de retourner directement la *SqlConnection* (ou *OracleConnection*, *OleDbConnection*).


Voici un exemple de ce que pourrait être votre PM :

```
public IDbCommand CreateCommand(connectionType type, string commandText)
{
    // Ajouter la gestion des cas d'erreur
    return CreateConnection(type).CreateCommand();
}
```

Je ne vais pas trop détailler les différentes interfaces, vous avez saisi le principe, je vais simplement rappeler quelques méthodes utiles :

```
// Crée un DataReader
IDataReader reader = maCommande.ExecuteReader();
...
// Crée un IDataParameter pour une requête
IDataParameter param = maCommande.CreateParameter();
```

Pour plus d'informations la documentation du framework contient tout ce dont vous avez besoin.

 *Même si cette question concerne principalement le framework 1.1, l'approche utilisée est applicable pour les versions supérieures du framework. Cependant à partir de la version 2.0, nous avons à notre disposition des outils plus puissants. Nous allons tout de suite en parler.*

III - Abstraction des classes en .NET 2.0

Nous venons de voir de quelle manière écrire du code générique en .NET 1.1 en utilisant les interfaces du namespace *System.Data*. A partir du framework 2.0, un namespace a été considérablement mis à jour : *System.Data.Common*. En version 1.1, ce namespace ne contenait que quelques classes parmi lesquelles *DataAdapter*, pour ne citer que la plus utilisée. Voyons sans plus attendre les nouveautés de *System.Data.Common* en .NET 2.0.

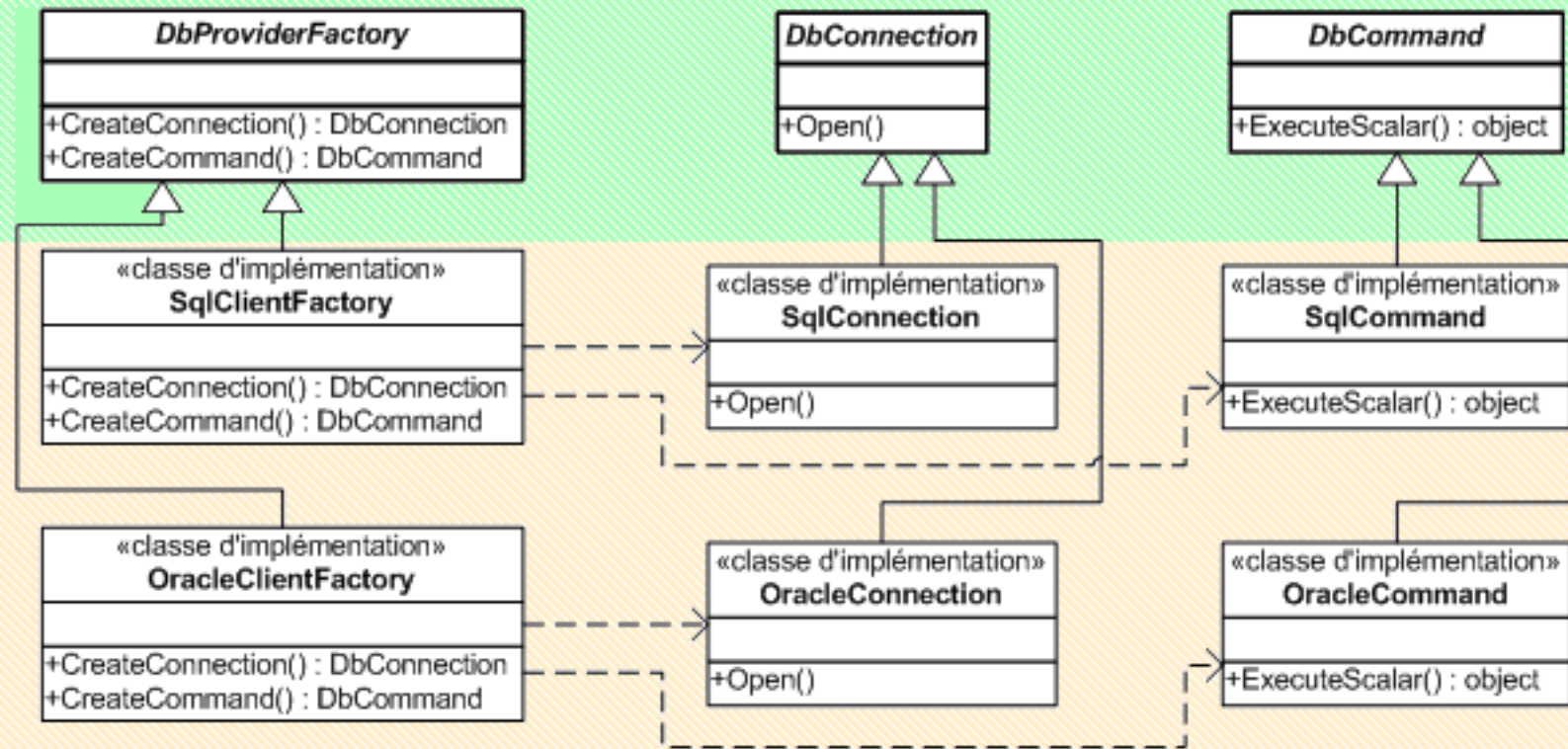
En faisant un tour dans la MSDN, on peut avoir une vue d'ensemble des classes ajoutées. La majorité des classes est nommée *Db[quelquechose]*, ceci donne déjà une vague idée de leur utilité, en un mot, cela concerne les bases de données (DB), mais ne donne pas d'indication sur le SGBD en question (à l'inverse des *OracleConnection* ou *SqlCommand* auxquels nous sommes habitués).

Toutes ces classes sont en effet des classes génériques destinées à étendre l'utilisation des interfaces. La première chose à remarquer est que les classes génériques implémentent ces mêmes interfaces, ce qui explique que le code écrit dans la section précédente fonctionne en l'état en .NET 2.0, ce qui est une bonne chose lors des migrations vers le framework 2.0.

La principale nouveauté concerne la notion de "provider" (fournisseur). Cette notion est matérialisée par les classes *DbProviderFactories* et *DbProviderFactory*.

Pour la petite information, *DbProviderFactories* est une "usine abstraite" ou "Abstract Factory" (en anglais, ça donne l'air de mieux s'y connaître). Il s'agit d'un design pattern, c'est en quelque sorte une super Factory. L'idée de l'abstract factory est d'encapsuler un groupe de factory ayant un thème commun. L'objectif dans le cas de *DbProviderFactory* est de pouvoir obtenir une factory dépendante de la source de données sans pour autant devoir connaître le type même de cette factory.

Classes utilisées par les clients



Classes d'implémentation utilisées en interne

Illustration 1: Schéma de l'abstract factory *DbProviderFactory*

DbProviderFactories permet d'énumérer les fournisseurs disponibles pour l'application (ex : SQL Server, Oracle, MySQL, SQL Server Compact Edition, etc). Elle est aussi responsable de la création des générateurs liés à une source de données spécifique.

DbProviderFactory représente un générateur d'objets liés à un fournisseur donné. La classe contient des méthodes pour créer des *DbCommand*, *DbConnection*, *DbParameter*. Ces objets sont dépendant du fournisseur, cela signifie que la méthode *CreateCommand* va, en fonction de la source de données, instancier un objet *DbCommand* avec une *SqlCommand* ou une *OracleCommand*. C'est exactement le même fonctionnement que les classes que nous avons créées pour le framework 2.0, sauf que cette fois c'est déjà prévu dans le framework.

Passons tout de suite à une explication plus pratique. Je vous ai dit juste avant que *DbProviderFactories* permet d'énumérer les fournisseurs disponibles, en réalité, cette classe va lire les paramètres définis dans le fichier `machine.config`. Ce dernier contient dans la section `<system.data>` une balise `<DbProviderFactories>` qui définit un par un les fournisseurs installés.

Exemple :

```
<system.data>
```

```
<DbProviderFactories>
  <add name="SqlClient Data Provider"
    invariant="System.Data.SqlClient"
    support="FF"
    description=".Net Framework Data Provider for SqlServer"
    type="System.Data.SqlClient.SqlClientFactory, System.Data,
      Version=2.0.3600.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089" />
  <add name="Odbc Data Provider"
    invariant="System.Data.Odbc"
    support="BF" description=
      ".Net Framework Data Provider for Odbc"
    type="System.Data.Odbc.OdbcFactory, System.Data,
      Version=2.0.3600.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089" />
  ... etc ...
</DbProviderFactories>
</system.data>
```

Pour récupérer cette liste, il suffit d'appeler la méthode *DbProviderFactories.GetFactoryClasses()* qui renvoie une *DataTable* contenant les données des fournisseurs. Cela vous permet de savoir dans votre application quelles sont les bibliothèques d'accès aux bases de données disponibles à l'exécution. Vous pouvez ensuite instancier la factory associée au provider en utilisant la méthode *DbProviderFactories.GetFactory(...)*, cette méthode prend en paramètre le nom du provider (*System.Data.SqlClient* par exemple) ou bien une *DataRow* issue du *DataTable* précédemment cité.

Une fois votre *DbProviderFactory* créé, vous pouvez utiliser ses méthodes *CreateConnection*, *CreateCommand* etc pour obtenir des instances des classes liées à la source de données.

Ceci n'est pas le plus utile pour la plupart des développeurs. En réalité, cette fonctionnalité est surtout utile pour les développeurs d'add-ins et de bibliothèques orientées "accès aux données". Le plus important pour le développeur est la possibilité d'utiliser les *DbProviderFactory* pour écrire un code générique. Il permet de remplacer les interfaces utilisées dans l'exemple précédent par des classes abstraites. Le principal avantage est que les classes génériques *Db** permettent de gérer tous les types d'accès aux données : mode connecté et déconnecté. Ce n'était pas possible avec les interfaces *IDb** car il n'existait pas de *DataAdapter* générique.

Voici un exemple de ce à quoi pourrait ressembler votre code :

```
public class DataProvider
{
    private static DbProviderFactory factory =
    DbProviderFactories.GetFactory("System.Data.SqlClient");

    public static DbConnection CreateConnection()
    {
        DbConnection connection = factory.CreateConnection();
        connection.ConnectionString =
        ConfigurationManager.ConnectionStrings["MyConnectionString"].ConnectionString;
        connection.Open();

        return connection;
    }

    public static DbCommand CreateCommand(string commandText)
    {
        DbCommand command = CreateConnection().CreateCommand();
        command.CommandText = commandText;

        return command;
    }
}
```

```
public static DbDataReader CreateDataReader(string commandText)
{
    return
CreateCommand(commandText).ExecuteReader(System.Data.CommandBehavior.CloseConnection);
}

public static DbDataReader CreateDataReader(DbCommand command)
{
    return command.ExecuteReader(System.Data.CommandBehavior.CloseConnection);
}

public static DataSet CreateDataSet(string commandText)
{
    DbDataAdapter adapter = factory.CreateDataAdapter();
    adapter.SelectCommand = CreateCommand(commandText);

    DataSet ds = new DataSet();
    adapter.Fill(ds);
    adapter.SelectCommand.Connection.Close();

    return ds;
}
}
```

Ce code n'est qu'un très simple exemple de ce qu'il est possible de faire avec les classes génériques. A la différence de l'exemple en .NET 1.1, celui ci se focalise sur l'écriture d'un code non dépendant de la source de données, alors que le premier exemple mettait l'accent sur l'utilisation de plusieurs sources de données de manière quasi transparente.

En utilisant ces techniques, vous êtes maintenant capables d'écrire un code non dépendant de la source de données et/ou apte à gérer plusieurs sources de données de type différent en parallèle.

Passons maintenant à une autre bonne pratique en ce qui concerne l'interrogation d'une base de données. Cette fois ci, cela ne concerne pas le type de la source de données mais plutôt la manière de l'interroger.

IV - Utilisation de requêtes paramétrées

La plupart d'entre vous construit les requêtes SQL en les concaténant morceau par morceau en insérant un morceau de SELECT par-ci, un morceau de FROM par-là et encore un bout de WHERE au passage. La plupart s'en satisfait ainsi, ne se posant pas la question de savoir s'il agit de la bonne méthode, et encore moins s'il en existe une autre. Et dans la plupart des cas, ça fonctionne, enfin disons plutôt, ça ne plante pas trop.

Alors vous avez peut-être déjà entendu quelqu'un parler de ce que l'on appelle les requêtes paramétrées (RP). Pour mettre les choses au point tout de suite, les RPs existent depuis longtemps, bien longtemps, alors que .NET n'était pas près de voir le jour. Il s'agit d'une fonctionnalité peu utilisée voire méconnue de la plupart des gens (sauf les DBA, en considérant en plus que les DBA soient des gens, mais c'est un autre problème).

IV-A - Le problème

Avant d'étudier ce qu'est réellement une RP en .NET, attardons nous sur une requête classique écrite par un développeur lambda. L'objectif de la requête est d'extraire de la table commandes, les identifiants des commandes enregistrées par un certain vendeur depuis une certaine date. Voici comment se présente la requête dans la plupart des cas :

```
SELECT id FROM commandes WHERE vendeur = [NOM] AND date > [DATE]
```

Les valeurs entre crochets indiquent les endroits où il faut insérer une valeur spécifique. Ce qui donne dans la plupart des cas lors de la première écriture :

```
string nom = "Georges";
DateTime date = DateTime.Today.AddDays(-7);
IDbCommand maCommande = maConnexion.CreateCommand();
maCommande.CommandText = "SELECT id FROM command WHERE login=" + nom +
    " AND date > " + date.ToString("dd/MM/yyyy HH:mm:ss");
```

Première exécution de la requête, ça explose. La raison, le format de la date, il y a un espace au milieu. Etant spécialiste SQL, vous ajoutez des apostrophes autour de la date, et vous transformez l'instruction en :

```
string nom = "Georges";
DateTime date = DateTime.Today.AddDays(-7);
IDbCommand maCommande = maConnexion.CreateCommand();
maCommande.CommandText = "SELECT id FROM command WHERE login='" + nom +
    "' AND date > '" + date.ToString("dd/MM/yyyy HH:mm:ss") + "'";
```

Deuxième exécution, re-plantage, cette fois ci, c'est le format de la date qui est en cause. En effet le serveur est dans une culture anglo-saxonne et le format de la date est mois/jour/année. Qu'à cela ne tienne :

```
string nom = "Georges";
DateTime date = DateTime.Today.AddDays(-7);
IDbCommand maCommande = maConnexion.CreateCommand();
maCommande.CommandText = "SELECT id FROM command WHERE login='" + nom +
    "' AND date > '" + date.ToString("yyyy-MM-dd") + "'";
```

Cette fois, la date fonctionne plus ou moins, mais ça plante encore. Vous aurez remarqué que j'en ai profité pour changer le nom car, problème, tout le monde ne s'appelle pas Georges. Et dans ce cas, notre vendeur est Irlandais (pourquoi pas), et il s'appelle O'Hara. Vous vous doutez déjà de ce qui s'est passé, l'apostrophe dans le nom a encore fait exploser la requête, nous avons une droit à une belle erreur de syntaxe.

Vous avez tous déjà eu ce problème (pas les Irlandais, mais les apostrophes). Encore une fois, une nouvelle correction, vous êtes toujours un pro de SQL et vous avez tout de suite pensé "je vais 'échapper' les apostrophes". Soit :

```
string nom = "O'Hara";
DateTime date = DateTime.Today.AddDays(-7);
IDbCommand maCommande = maConnexion.CreateCommand();
maCommande.CommandText = "SELECT id FROM command WHERE login='" +
    nom.Replace("'", "'') +
    "' AND date > '" + date.ToString("yyyy-MM-dd") + "'";
```

Victoire, cette fois, ça fonctionne. Enfin, j'ai envie de dire que ça ne plante plus, et encore ça me semble optimiste. Si vous avez un tant soit peu d'amour propre, vous avez honte de ce code, et je dois avouer que je vous comprends. Il doit sûrement y avoir une meilleure solution. Finalement, une utilité pourrait être trouvée pour ces RP. Vous avez eu raison de garder espoir.

IV-B - La solution

Concrètement, qu'est ce qu'une requête paramétrée ? Il s'agit d'une requête qui contient des variables à la place de certaines valeurs. Vous pouvez sans problème écrire une requête paramétrée dans votre éditeur SQL préféré.

Voici une requête exemple :

```
SELECT id FROM commandes WHERE vendeur = @p_name AND date > @date_effet
```

Cette requête extrait de la table commandes, les identifiants des commandes enregistrées par un certain vendeur depuis une certaine date. Dans cet exemple, les deux valeurs pour le login et le mot de passe sont remplacées par deux paramètres identifiés par l'@ qui les précède. En fonction de votre navigateur SQL (*TOAD*, *SQLNavigator*, *Entreprise Manager*, *SQL Server Management Studio Express*, etc), vous serez peut-être invités à saisir des valeurs pour l'exécution de la requête, dans le cas contraire, la requête va sûrement échouer car les paramètres sont en fait des variables qui doivent être déclarées plus haut dans le script (ceci n'est valable que pour l'exécution dans un éditeur SQL).

Passons maintenant à l'écriture de ce type de requête en utilisant le framework .NET. L'écriture se fait comme pour une requête traditionnelle :

```
IDbCommand maCommande = maConnexion.CreateCommand();
maCommande.CommandType = CommandType.Text;
maCommande.CommandText = "SELECT id FROM command WHERE login = @login AND date > @date_effet";
Il suffit ensuite de déclarer deux magnifiques paramètres SQL comme ceci :
IDbParameter paramNom = maCommande.CreateParameter();
paramNom.Name = "@login";
paramNom.DbType = DbType.VarChar;
paramNom.Direction = ParameterDirection.Input;
paramNom.Value = "O'Hara";

IDbParameter paramDate = maCommande.CreateParameter();
paramDate.Name = "@date_effet";
```

```
paramDate.DbType = DbType.DateTime;
paramDate.Direction = ParameterDirection.Input;
paramDate.Value = DateTime.Today.AddDays(-7).Date;
```

On termine par l'ajout de ces paramètres dans la liste des paramètres de la commande :

```
maCommande.Parameters.Add(paramNom);
maCommande.Parameters.Add(paramDate);
```

Il ne reste plus qu'à exécuter votre requête de manière habituelle.

Pour résumer, nous venons de :

- a Créer une requête avec des "blancs", les paramètres,
- b Créer des paramètres SQL dont le nom correspond au nom d'un des paramètres,
- c Affecter un type et une valeur correspondante à chaque paramètre,
- d Associer les paramètres à notre requête,
- e Exécuter la requête sur la source de données sans se ramasser un tas d'erreurs et sans se fatiguer,
- f Avoir honte en repensant à toutes les bidouilles mises en place avant pour éviter tous ces problèmes.

L'intérêt des paramètres ne se manifeste pas seulement dans le scénario précédemment décrit. En effet les paramètres SQL ne sont pas un concept de développement, mais un concept de base de données. Il existe des raisons supplémentaires d'utiliser des paramètres SQL qui sont cette fois plus liées à la source de données. Pour cela intéressons nous quelque peu à la manière de travailler de nos SGBD préférés (même s'il existe des différences entre les différents produits, l'idée est la même à chaque fois).

Lors de l'envoi d'une requête au SGBD, la requête est d'abord analysée pour en extraire les paramètres par exemple, mais aussi les index qui vont être utilisés, etc. Ces données recueillies sont ensuite placées (avec la requête) dans un cache interne. De cette manière, si la même requête revient, il n'y aura pas nécessité de la ré-analyser. Ce système fonctionne bien SAUF dans le cas où toutes les requêtes sont différentes, dans ce cas, le cache va grossir inutilement, les requêtes vont être analysées à chaque fois. Vous êtes en train de vous dire " oh mon dieu, c'est horrible ", et vous avez raison. Vous pensez aussi " Ouf, heureusement que j'utilise souvent la même requête, je ne tombe jamais dans ce cas-là ", et là, c'est le drame, vous vous trompez lamentablement ! Car oui, vous êtes en fait dans ce cas horrible qui fait peur au DBA. " Tu mens " me direz vous, et vous allez même ajouter " quand je fais des insertions en masse, c'est le même INSERT à chaque fois ". Et bien non, je ne mens pas et je le prouve. Quand vous faites vos insertions, vous utilisez sûrement une requête de type :

```
while (jAiDesDonneesEnAttente)
{
    IDbCommand maCommande = maConnexion.CreateCommand();
    maCommande.CommandType = CommandType.Text;
    maCommande.CommandText = "INSERT INTO maTable VALUES(' " + maDonneeQuiChangeAChaqueIteration +
        "' )";
    maCommande.ExecuteNonQuery();
}
Au final, vous envoyez au SGBD des requêtes du genre :
"INSERT INTO maTable VALUES('TOTO') "
"INSERT INTO maTable VALUES('NENESS') "
"INSERT INTO maTable VALUES('JUNIOR') "
```

Ce sont, vous l'avez maintenant compris, des requêtes complètement différentes du point de vue du SGBD.

Dans le cas présent, si vous aviez utilisé un paramètre SQL pour la valeur à enregistrer, la requête aurait été la même à chaque fois, et uniquement la valeur du paramètre aurait été modifiée. Vous pouviez même écrire tout le code de création de la requête en dehors de la boucle pour ne laisser dans celle-ci que :

```
while ( jAiDesDonneesEnAttente )
{
    monParametre.Value = maDonneeQuiChangeAChaqueIteration;
    maCommande.ExecuteNonQuery();
}
```

C'est finalement plus lisible/robuste/efficace que la première méthode.

Abordons maintenant un autre avantage inestimable des paramètres SQL, vous vous mettez à l'abri des injections SQL. Il s'agit d'une attaque courante, principalement pour les applications Web, qui consiste à introduire dans un champ de saisie, une chaîne représentant un morceau de requête SQL qui, si l'application n'est pas protégée, va être exécuté dans la source de données à votre insu. Pour illustrer, reprenons le code précédent en le modifiant un peu :

```
IDbCommand maCommande = maConnexion.CreateCommand();
maCommande.CommandType = CommandType.Text;
maCommande.CommandText = "INSERT INTO maTable VALUES('" + monTextBox.Text + "')";
maCommande.ExecuteNonQuery();
```

Les utilisateurs sympas vont saisir des chaînes gentilles comme " bonjour " ou " mon PC il é KC ". Ces utilisateurs là sont comme la pitié dans l'oeil d'un percepteur d'impôt, ils n'existent pas. Non à la place, vous aurez des utilisateurs mal rasés avec les cheveux longs (des méchants utilisateurs) qui vont essayer de saisir des chaînes du style " '); DROP DATABASE; -- ". Au final, votre requête va donner : INSERT INTO maTable VALUES(""); DROP DATABASE; --')

Au revoir la base de données, merci d'avoir joué. Si vous aviez utilisé un paramètre SQL, la chaîne passée dans la requête aurait été 'échappée' et au final, vous auriez inséré sans encombre la valeur saisie dans la zone de texte. J'en vois un paquet qui sont en train de paniquer, oui, effectivement, il est temps d'aller modifier vos applications avant qu'il ne soit trop tard. Voilà pour la partie SQL Injection.

Encore un autre avantage, en concaténant vos valeurs dans la requête, vous introduisez une source d'erreur, une couche d'illisibilité supplémentaire, car vous êtes toujours obligé de compter les apostrophes, de savoir où il faut ajouter les plus, les virgules, les parenthèses qui manquent, etc. En utilisant les paramètres, vous avez la possibilité de créer la requête et de la tester dans un outil externe (Entreprise Manager par exemple), et de la copier/coller en l'état sans aucune modification dans votre code. Gain de temps en développement et en maintenance future.

Et un petit dernier pour la route, en concaténant la requête, vous passez à côté de toute la belle gestion des erreurs du framework en abandonnant une hypothétique *InvalidCastException* en débogage au 'profit' d'une espèce d'erreur SQL générique qui ne vous donne aux mieux qu'une appréciation minable de la source de l'erreur. En effet en cas d'erreur sur un paramètre SQL, vous obtenez des exceptions typées comme des *InvalidCastException* ou autre, alors qu'en utilisant une requête "concaténée", vous n'avez que des exceptions génériques... Pas terrible pour le débogage, non ?

Vous avez donc maintenant une solide connaissance de l'(immense) intérêt des paramètres SQL. Il ne vous reste plus qu'à mettre tout ça en pratique.

V - Conclusion

Nous avons vu au travers de ces différents points qu'en matière d'accès aux données, vous avez à votre disposition toute une série d'outils qui vous permettent de gagner du temps de débogage et de maintenance. Il vous reste maintenant à mettre en pratique tous ces concepts pour qu'ils deviennent des automatismes, car ce qui différencie un développeur standard et un bon développeur n'est pas la rapidité de développement, mais le rapport entre la qualité du code et le temps de développement.

VI - Remerciements

Je pourrais jamais assez remercier **Aspic** pour toute l'aide qu'il m'a apportée pour la rédaction de cet article. Un autre grand merci à **caro95470** pour la correction orthographique de cet article.

