

# Utiliser le modèle ADOX avec Visual Basic

Ecrit par J-M RABILLOUD de [www.developpez.com](http://www.developpez.com). Reproduction, copie, publication sur un autre site Web interdite sans autorisation de l'auteur.

## Remerciements

J'adresse ici tous mes remerciements à l'équipe de rédaction de "developpez.com" et tout particulièrement à Etienne Bar, Sébastien Curutchet, Maxence Hubiche et Romain Puyfoulhoux pour le temps qu'ils ont bien voulu passer à la correction et à l'amélioration de cet article.

<b>Introduction</b>	<b>3</b>
<b>Préambule</b>	<b>3</b>
<b>Rappels Access</b>	<b>4</b>
<b>Sécurité</b>	<b>4</b>
<b>Paramétrage JET</b>	<b>4</b>
<b>Rappels ADO</b>	<b>4</b>
<b>Propriétés statiques &amp; dynamiques (Properties)</b>	<b>4</b>
<b>L'objet Connection</b>	<b>5</b>
<b>L'objet Command</b>	<b>6</b>
Généralités	6
Propriétés	6
Méthodes	7
Collection Properties	7
Collection Parameters	8
Objet Parameter	8
Exemple	9
<b>OpenSchema</b>	<b>11</b>
<b>Modèle objet</b>	<b>12</b>
<b>Notions Fondamentales</b>	<b>12</b>
<b>ADOX &amp; Access</b>	<b>12</b>
<b>Propriétaire</b>	<b>13</b>
<b>ParentCatalog</b>	<b>13</b>
<b>L'objet Catalog</b>	<b>13</b>
<b>Collections de l'objet Catalog</b>	<b>14</b>
<b>Collection tables</b>	<b>14</b>
<b>Collection Procedures</b>	<b>14</b>
<b>Collection Views</b>	<b>14</b>
<b>Collection Groups</b>	<b>14</b>
<b>Collection Users</b>	<b>15</b>
<b>L'objet Table</b>	<b>15</b>
<b>Collection Properties</b>	<b>15</b>
<b>Collection Columns</b>	<b>16</b>
Objet Column	16
<b>Collection Indexes</b>	<b>18</b>

Objet Index	18
<b>Collection Keys</b>	<b>19</b>
Quelques notions	19
Méthode Append	19
Objet Key	20
Exemple	21
<b>Conclusion sur les tables</b>	<b>22</b>
<b><i>L'objet Procedure</i></b>	<b>22</b>
<b>Création d'un objet procédure</b>	<b>22</b>
Pas d'objet Parameter	22
<b>Modification d'un objet Procedure</b>	<b>23</b>
<b><i>L'objet View</i></b>	<b>23</b>
<b>Conclusion sur les objets View &amp; Procedure</b>	<b>24</b>
<b><i>Gestion des utilisateurs</i></b>	<b>25</b>
<b>Cas particulier d'Access</b>	<b>25</b>
<b>Propriétés et droits</b>	<b>25</b>
Propriétaire	25
Administrateur	26
Utilisateurs et groupes	26
Héritage des objets	26
<b>Objet Group</b>	<b>27</b>
SetPermissions	27
GetPermissions	29
<b>Objet User</b>	<b>30</b>
ChangePassword	30
GetPermissions & SetPermissions	30
Properties	30
<b>Exemple</b>	<b>30</b>
<b>Techniques de sécurisation</b>	<b>31</b>
Modification	32
Création	32
Une autre solution : le DDL	32
<b>Conclusion sur la sécurité</b>	<b>33</b>
<b><i>Conclusion</i></b>	<b>33</b>

# Introduction

Le présent article va aborder la programmation du modèle objet ADOX (Microsoft® ActiveX® Data Objects Extensions) pour le Langage de Définition des Données (*DDL = Data Definition Language and Security*).

Dès à présent, il faut bien comprendre que la bibliothèque ADOX sert pour la création ou la modification d'une base de données. Pour une simple consultation, il est beaucoup plus efficace d'utiliser la méthode OpenSchema d'ADO. Toutefois, à des fins explicatives, vous trouverez du code "de consultation" dans cet article.

Ce qui était du temps de DAO dans l'objet Database, c'est à dire la structure de la base (table, vue...) et dans l'objet Workspace (utilisateurs, groupes) se trouve réuni dans le modèle objet ADOX. Avant de pouvoir l'utiliser, il faut référencer la bibliothèque "Microsoft ADO Ext. 2.x for DDL and Security"

Il n'est pas toujours évident de voir la limite entre les modèles ADO et ADOX. Pour simplifier, on utilise ADOX pour créer ou modifier un élément de la base de données, et ADO pour manipuler les données. De plus ADOX est le seul moyen de gérer les utilisateurs et les groupes.

Dans la suite de cet article, après avoir vu quelques généralités, nous allons parcourir le modèle objet tout en voyant les pièges et astuces qui le composent, avec des exemples de création d'une base de données afin de bien comprendre son utilisation.

**N.B :** Comme nous le verrons par la suite, de nombreuses propriétés dépendent du fournisseur de données. Donc, il vous appartiendra de vous renseigner sur les capacités du fournisseur qui vous intéresse.

## [Fondamentaux ADOX \(Microsoft\)](#)

Si vous ne l'avez pas déjà fait je vous invite aussi à lire :

- ➔ [Accès aux bases de données "ADO" avec Visual Basic 6.0](#)
- ➔ [tutoriel ADO pour VB6](#)

Remarque : Les exemples de cet article ont été faits avec une base Access 2000 et ADOX version 2.7.

**Attention :** Le fournisseur Jet 3.51 ignore un grand nombre des fonctionnalités exposées dans cet article, notamment sur la sécurité.

Si vous avez des questions à poser, venez sur les forums de <http://www.developpez.com/>

- ➔ [Forum Visual Basic](#)
- ➔ [Forum Access](#)

En respectant [les règles du forum](#).

## **Préambule**

La quasi-totalité des exemples de cet article porte sur la programmation d'une base Access. Pourtant on peut noter que :

Pour la programmation des bases Access DAO est un modèle beaucoup plus complet qu'ADO. ADO est plutôt orienté SQL-Server.

J'ai fais ce choix car fondamentalement les différences sont uniquement dues aux fournisseurs et que le fournisseur Jet est le plus complexe à programmer. Partant du principe que les utilisateurs Access sont nombreux à utiliser ADO depuis la version 2000, j'ai écrit cet article afin qu'ils puissent comprendre les problèmes qu'ils rencontrent. Pour les utilisateurs d'autres SGBD, les problèmes ne seront que moins nombreux.

Bonne lecture.

# Rappels Access

## **Sécurité**

Il y a deux types de sécurité Access : la sécurité au niveau partage, c'est à dire l'utilisation d'un mot de passe pour ouvrir la base de données et la sécurité au niveau utilisateur semblable aux autres SGBD. Celle-ci repose sur l'identification de l'utilisateur, celui-ci ayant un certain nombre de droits définis. La différence d'Access vient de l'utilisation d'un fichier externe pour gérer cette sécurité. Les deux niveaux de sécurité sont cumulables.

## **Paramétrage JET**

Par défaut, le paramétrage du moteur Jet se trouve dans la base de registre à la clé :  
HKLM\Software\Microsoft\Jet\4.0\Engines\Jet 4.0.

Ce paramétrage pouvant être différent d'un poste à l'autre, il convient d'utiliser le code pour toujours mettre ce paramétrage aux valeurs désirées. Pour cela deux approches sont possibles : la modification des valeurs du registre ou le paramétrage de la connexion (méthode que nous verrons plus loin).

# Rappels ADO

Sans que cet article couvre le modèle ADO, nous allons voir ici quelques notions qui nous seront utiles pour l'utilisation d'ADOX ou la récupération de schéma.

## **Propriétés statiques & dynamiques (Properties)**

Dans le modèle ADO et a fortiori dans le modèle ADOX, de nombreux objets possèdent une collection un peu particulière : "Properties". Celle ci concerne des propriétés dites dynamiques par opposition aux propriétés habituelles des objets (statiques). Ces propriétés dépendent du fournisseur de données, elles ne sont en général accessibles qu'après la création de l'objet (et éventuellement l'application d'une méthode refresh sur la collection) voire après l'ouverture de l'objet.

On ne peut pas accéder à une propriété statique par l'intermédiaire de la collection Properties.

Un objet **Property** dynamique comporte quatre propriétés intégrées qui lui sont propres, à savoir :

- La propriété **Name** qui est une chaîne identifiant la propriété
- La propriété **Type** qui est un entier spécifiant le type de donnée de la propriété.
- La propriété **Value** qui est un variant contenant la valeur de la propriété.
- La propriété **Attributes** qui est une valeur de type Long indiquant les caractéristiques de propriétés spécifiques au fournisseur.

## L'objet Connection

Dans son utilisation la plus courante la chaîne de connexion prend deux paramètres : le fournisseur et la source de données. Pourtant l'objet Connection comprend de nombreuses propriétés permettant de gérer le mode d'accès, la sécurité, le paramétrage Jet, etc...

Pour mieux appréhender l'objet Connection et ses propriétés, regardons le code suivant.

```
Dim cnn1 As ADODB.Connection

Set cnn1 = New ADODB.Connection
With cnn1
Debug.Print .Properties.Count
.Provider = "Microsoft.Jet.OLEDB.4.0;"
.ConnectionTimeout = 30
.CursorLocation = adUseClient
.Mode = adModeShareExclusive
.Properties("Jet OLEDB:System database") =
"D:\User\jmarc\tutorial\ADOX\system.mdw"
Debug.Print .Properties.Count
.Open "Data Source=D:\User\jmarc\tutorial\ADOX\baseheb.mdb ;User Id=Admin;
Password="
Debug.Print .Properties.Count
End With
```

Si vous exécutez ce code en l'état, vous allez avoir l'erreur "Le fournisseur indiqué est différent de celui déjà utilisé" [3220 – adErrCantChangeProvider]

Ceci vient du fait que l'appel de Properties.Count crée l'instance de l'objet et non le Set comme une erreur courante le fait penser. Or comme à ce moment le fournisseur n'a pas été défini, ADO utilise le fournisseur par défaut "MSDASQL.1". Dès lors, il n'est plus possible de modifier le fournisseur ce qui déclenche une erreur.

**Règle n°1** : On doit toujours définir la propriété Provider en premier sur un objet Connection.

On enlève donc le premier Debug.Print et on exécute le code. On observe le résultat dans la fenêtre correction et on constate qu'il y a 27 propriétés avant l'ouverture et 94 après.

**Règle n°2** : On valorise les propriétés statiques et les propriétés dynamiques dépendantes du fournisseur avant l'ouverture de la connexion.

En fait les propriétés dépendantes du fournisseur qu'il nous faut valoriser avant l'ouverture sont celles qui gèrent la sécurité (voir le chapitre correspondant).

Stricto sensu, on peut valoriser toutes ces propriétés lors de l'ouverture de la connexion, mais pour des raisons de maintenance (lisibilité) il est souvent utile de décomposer la connexion avant l'ouverture. La chaîne de connexion à passer dans ce cas serait :

```
Cnn1.ConnectionString="Provider=Microsoft.Jet.OLEDB.4.0;Password="";User
ID=Admin;Data Source=D:\User\jmarc\tutorial\ADOX\baseheb.mdb;Mode=Share
Deny Read|Share Deny Write;Extended Properties="";Jet OLEDB:System
database=D:\User\jmarc\tutorial\ADOX\system.mdw;Jet OLEDB:Registry
Path="";Jet OLEDB:Database Password="";Jet OLEDB:Engine Type=5;Jet
OLEDB:Database Locking Mode=0;Jet OLEDB:Global Partial Bulk Ops=2;Jet
OLEDB:Global Bulk Transactions=1;Jet OLEDB:New Database Password="";Jet
OLEDB:Create System Database=False;Jet OLEDB:Encrypt Database=False;Jet
OLEDB:Don't Copy Locale on Compact=False;Jet OLEDB:Compact Without Replica
Repair=False;Jet OLEDB:SFP=False"
```

**Règle n°3** : Lorsqu'on doit valoriser plusieurs propriétés de la connexion, on le fait en dehors de la chaîne de connexion.

Il y aurait encore de nombreuses choses à dire sur l'objet Connection, mais cela sort du cadre de cet article.

## L'objet Command

Beaucoup de programmeurs n'utilisent pas l'objet Command avec ADO préférant travailler directement avec l'objet Recordset. Dans le modèle ADOX, l'objet Command est indispensable. Nous allons donc aborder l'objet Command de façon assez détaillée sous l'angle suivant:

- ❖ Utilisation de Command pour créer des requêtes stockées dans la base de données (paramétrées ou non)

Sachez toutefois que l'objet Command s'utilise dans de nombreuses autres situations. Nous allons commencer par voir l'objet Command en détail.

### Généralités

Un objet command représente une commande spécifique à exécuter sur une source de données. Ceci peut être une instruction SQL ou une requête/procédure stockée dans la base ou créée à l'exécution. Un objet Command dépend toujours d'une connexion soit créée spécifiquement pour cette commande, soit d'une connexion déjà existante.

### Propriétés

#### ActiveConnection

Définit la connexion utilisée pour l'objet Command. Cette propriété doit être passée à Nothing avant de changer la connexion d'un objet Command. Comme les objets Command héritent de certaines propriétés de la connexion, faites attention au paramétrage de celle-ci.

#### CommandText & CommandStream

Nous n'étudierons pas dans cet article la programmation de l'objet Stream. Sachez toutefois qu'il est indispensable pour la programmation Internet et l'utilisation du XML. Ces deux propriétés sont exclusives. La propriété CommandText contient le texte de la commande à exécuter. Ce texte peut être le nom d'une procédure stockée, une chaîne SQL etc...

#### CommandTimeout

La valeur est en secondes. S'applique en général sur l'objet Connection

#### CommandType

Donne le type de la commande. Cette propriété est très importante. En effet, il y aura une erreur récupérable si le paramètre donné est faux. De plus ne pas valoriser correctement cette propriété peut dégrader fortement les performances. Les valeurs peuvent être :

Constante	Description
adCmdText	CommandText correspond à la définition textuelle d'une commande ou d'un appel de procédure stockée
adCmdTable	CommandText correspond au nom de table dont les colonnes sont toutes renvoyées par une requête SQL générée en interne.
adCmdTableDirect	CommandText correspond à un nom de table dont les colonnes sont toutes renvoyées.
adCmdStoredProc	CommandText correspond au nom d'une procédure stockée.
adCmdUnknown	Valeur utilisée par défaut. Le type de commande de la propriété CommandText est inconnu
adCmdFile	CommandText correspond au nom de fichier d'un Recordset permanent
adExecuteNoRecords	CommandText correspond à une commande ou une procédure stockée qui en renvoie pas de ligne (par exemple, une commande qui insère uniquement des données). Si des lignes sont extraites, elles ne sont pas prises en compte et ne sont pas retournées. Toujours associée à adCmdText ou adCmdStoredProc



Attention toutefois, une procédure stockée peut être différemment interprétée selon les SGBD. Voir aussi la rubrique "Parameters".

### Prepared

Détermine si une commande doit être Pré-compilée. N'est utile que si la commande doit être exécutée plusieurs fois. Attention, certains fournisseurs n'acceptent pas cette modification sans toutefois déclencher d'erreur.

### State

Renvoie l'état de la commande (ouverte ou fermée)

## **Méthodes**

### Cancel

Annule l'exécution de la commande si celle-ci est asynchrone.

### Execute

Exécute la commande. De la forme *command.Execute RecordsAffected, Parameters,*

### **Options**

#### **Remarques sur la méthode Execute**

Vous aurez noté qu'il existe une méthode Execute sur l'objet Connection et sur l'objet Command. Elles sont sensiblement identiques si ce n'est :

Une commande est réutilisable

Un objet Command est nécessaire pour les requêtes paramétrées et les procédures stockées.

Notez aussi qu'un recordset créé à l'aide de Connection.Execute hérite des propriétés de la connexion.

### CreateParameter

Sert à créer un paramètre (voir explications plus loin).

De la forme *command.CreateParameter (Name, Type, Direction, Size, Value)*

Attention à ne pas faire une erreur courante avec cette méthode. Celle-ci crée un paramètre mais ne l'ajoute pas à la collection Parameters de l'objet Command. Ceci est d'ailleurs indispensable afin de pouvoir valoriser d'autres propriétés avant l'ajout.

## **Collection Properties**

C'est la collection des propriétés dynamiques de l'objet Command. Comme il y en a beaucoup, nous n'allons pas toutes les voir en détail, mais certaines méritent que nous nous y arrêtions. Pourquoi?

Lors de l'utilisation d'un objet recordset, la définition des paramètres de celui-ci valorise un certain nombre de propriétés. Ceci n'est pas le cas sur un objet Command. En tout état de cause, ceci n'a pas d'importance si l'objet Command ne renvoie pas un recordset.

Bookmarkable	Boolean	Read/Write/Required	FAUX
Permet d'utiliser un Recordset qui accepte les signets.			
Jet OLEDB: Bulk Transactions	Integer	Read/Write/Required	0

Détermine si les requêtes action sont acceptées de façon partielle ou globale. De manière générale, les transactions partielles ne devaient jamais être autorisées.

Jet OLEDB: ODBC Pass-Through Statement	Boolean	Read/Write/Required	FAUX
Défini les requêtes SQL directes			

#### **Notion SQL Direct**

Les requêtes SQL direct étaient utilisées avec DAO pour écrire des requêtes n'ayant pas besoin d'être interprétées par Jet lors d'accès à des données externes. Comme une telle requête doit avoir une connexion vers la source de données, il n'est plus la peine de faire des requêtes SQL Direct.

Lock Mode	Integer	Read/Write/Required	1
-----------	---------	---------------------	---

Détermine dans une certaine mesure le mode de verrouillage. Cette propriété ne peut pas être dissociée de la propriété IsolationLevel

Others' Changes Visible	Boolean	Read/Write/Required	FAUX
Permet l'utilisation multi-utilisateurs			

## **Collection Parameters**

### **Généralités**

Les utilisateurs d'autres SGBD qu'Access veilleront à bien faire la différence entre procédure stockée et requête paramétrée.

Quoique regroupés dans la même collection, il existe deux types de paramètres. Les paramètres d'entrée, attendus par la procédure/requête pour pouvoir s'exécuter, et les paramètres de sortie qui peuvent être renvoyés par une procédure. Il convient de faire attention avec ceux-ci, une connexion n'acceptant jamais plus de deux objets Command ayant des paramètres de sortie (le paramètre de retour n'ayant pas d'influence).

### **Quelques méthodes de la collection.**

#### **Append**

Ajoute un paramètre à la collection. De la forme

**Command.Parameters.Append Name, Type, DefinedSize, Attrib**

Il est possible d'utiliser la méthode CreateParameter dans le Append. Vous devez avoir typé votre paramètre avant ou lors de l'ajout à la collection. Il est préférable de valoriser votre paramètre avant de l'ajouter à la collection afin de ne pas solliciter le fournisseur.

#### **Delete**

Enlève un paramètre de la collection. On doit spécifier soit le nom, soit l'index du paramètre à retirer.

## **Objet Parameter**

### **Propriétés**

#### **Attributes**

Précise si le paramètre accepte les valeurs signées, binaires ou NULL.

#### **Direction**

Définit si le paramètre est un paramètre d'entrée, de sortie ou de retour. Attention à ne pas confondre un paramètre de retour(adParamReturnValue) qui est l'entier éventuellement renvoyé par une procédure (avec l'instruction Return) d'un paramètre de sortie (adParamOutput) qui est le (ou les) paramètre renvoyé par une procédure (en général des valeurs tirées de la base de données).

#### **Name**

Définit le nom du paramètre. N'est pas obligatoire

#### **NumericScale**

Donne la précision (nombre de chiffres à droite de la virgule) d'un paramètre numérique.

#### **Size**

Donne la taille en octets d'un paramètre dont le type est potentiellement de longueur variable (par exemple String). Ce paramètre est obligatoire pour les types de longueur indéterminée (String, Variant).

Elle doit être toujours valorisée avant ou lors de l'ajout à la collection.

#### **Type**

Comme son nom l'indique !

Ne vous emmêlez pas les pinceaux entre les chaînes **adVarChar** et les chaînes Unicodes **adVarWChar**

#### **Value**

De manière générale, valorisez cette propriété **après** l'ajout à la collection.

### **Méthode**

#### **AppendChunk**

Permet d'accéder aux textes ou binaires longs.

## **Exemple**

### **Requêtes paramétrées**

Nous allons voir ici un exemple d'utilisation de requête paramétrée.

Prenons la requête paramétrée stockée dans la base nommée ReqFerie définie comme :

```
PARAMETERS DateCib DateTime;
SELECT tblFerie.DateFer
FROM tblFerie
WHERE (((tblFerie.DateFer)>DateValue([DateCib])))
ORDER BY tblFerie.DateFer;
```

La fonction suivante crée un recordset en utilisant l'objet Command.

```
Private Sub Command5_Click()
Dim cnn1 As ADODB.Connection, Comm1 As ADODB.Command, Param1 As Parameter,
MonRecordset As ADODB.Recordset

Set cnn1 = New ADODB.Connection
With cnn1
.Provider = "Microsoft.Jet.OLEDB.4.0;"
.ConnectionTimeout = 30
.CursorLocation = adUseClient
.IsolationLevel = adXactChaos
.Mode = adModeShareExclusive
.Properties("Jet OLEDB:System database") =
"D:\User\jmarc\tutorial\ADOX\system.mdw"
.Open "Data Source=D:\User\jmarc\tutorial\ADOX\baseheb.mdb ;User
Id=Admin; Password="
End With
Set Comm1 = New ADODB.Command
With Comm1
.ActiveConnection = cnn1
.CommandType = adCmdStoredProc
.CommandText = "ReqFerie"
End With
Set Param1 = New Parameter
With Param1
.Direction = adParamInput
.Type = adDate
.Name = "DateCib"
End With
Comm1.Parameters.Append Param1
Comm1("DateCib").Value = #1/1/2002#
Set MonRecordset = Comm1.Execute

End Sub
```

Notez que tel que ce code est écrit le curseur est statique en lecture seule. Ceci vient du fait qu'il s'agit d'une requête Select dans le vrai sens du terme est que celle-ci n'est pas modifiable.

### **DDL**

Je ne vais pas rentrer dans le détail du DDL dans cet article mais sachez qu'il est directement utilisable avec l'objet Command. Le code suivant créera une table avec un index:

```
With Comm1
.ActiveConnection = cnn1
.CommandType = adCmdText
.CommandText = "CREATE TABLE DeuxièmeTable(Prénom TEXT,Nom TEXT,
DateDeNaissance DATETIME,CONSTRAINT MaContrainteTable UNIQUE (Prénom, Nom,
DateDeNaissance)"
End With
Set MonRecordset = Comm1.Execute
```

### Procédure stockée

Imaginons la procédure stockée suivante (SQL-Server) :

```
CREATE PROCEDURE LoginValide
@UserName varchar(15),
@Password varchar(10)
As
if exists(Select * From TblUtilisateur
Where Nom = @UserName
And
MotDePasse = @Password)
return(1)
else
return(0) Go
```

Nous allons pouvoir l'utiliser de la façon suivante en visual basic.

```
Private Sub Command5_Click()
Dim cnn1 As ADODB.Connection, Comm1 As ADODB.Command, Param1 As Parameter,
MonRecordset As ADODB.Recordset
Dim compteur As Long

Set cnn1 = New ADODB.Connection
strCnn = "Provider=sqloledb;Data Source=svr;Initial Catalog=Hebdo;User
Id=sa;Password=; "
Set Comm1 = New ADODB.Command
With Comm1
.ActiveConnection = cnn1
.CommandType = adCmdStoredProc
.CommandText = "LoginValide"
End With
For compteur = 1 To 3
Set Param1 = New Parameter
With Param1
.Direction = Choose(compteur, adParamInput, adParamInput,
adParamReturnValue)
.Type = Choose(compteur, adVarChar, adVarChar, adInteger)
.Name = Choose(compteur, "Nom", "Passe", "Reponse")
.Size = Choose(compteur, 15, 10, 4)
End With
Comm1.Parameters.Append Param1
Set Param1 = Nothing
Next compteur
Comm1("Nom").Value = txtUser.Text
Comm1("Passe").Value = txtPassword.Text
Comm1.Execute
If Comm1("Reponse").Value = 0 Then Exit Sub
...
End Sub
```

La seule différence vient du fait qu'une procédure stockée peut renvoyer d'autres choses qu'un recordset. Dans notre exemple la procédure attend deux paramètres (nom et mot de passe) et renvoie 0 ou 1 selon l'existence de l'enregistrement dans la table.

Il faut que la réponse soit du type ReturnValue et non Output puisqu'elle est le résultat de l'instruction Return.

N.B : Vous retrouverez souvent la notation Choose dans cet article.

# OpenSchema

Comme je l'ai dit plus avant, il faut utiliser plutôt la méthode OpenSchema de l'objet Connection ADO pour récupérer des informations de structure. Cette méthode est de la forme suivante: **Set recordset = connection.OpenSchema (QueryType, Criteria, SchemaID)**

Où QueryType donne le type d'information à récupérer, Criteria définit les contraintes sur ces informations

SchemaID n'est utilisé que si QueryType est adSchemaProviderSpecific.

Pour avoir la liste complète des possibilités reportez-vous à l'aide MSDN (Méthode OpenSchema (ADO)).

Comme nous le voyons, les informations renvoyées se trouvent dans un recordset. Pour voir le principe de la méthode, observons le code suivant :

```
Private Sub RecupSchema()  
Dim cnn1 As ADODB.Connection, ReqTable As ADODB.Recordset, ReqChamp As  
ADODB.Recordset, compteur As Long  
Dim NomTab As String, fso As FileSystemObject, Fich As TextStream  
Set fso = New FileSystemObject  
Set Fich = fso.CreateTextFile("d:\Schemal.txt", True)  
Set cnn1 = New ADODB.Connection  
With cnn1  
    .Provider = "Microsoft.Jet.OLEDB.4.0;"  
    .Properties("Jet OLEDB:System database") = "D:\ADOX\system.mdw"  
    .Open "Data Source=D:\ADOX\baseheb.mdb ;User Id=Admin; Password=" &  
End With  
Set ReqTable = cnn1.OpenSchema(adSchemaTables)  
Do Until ReqTable.EOF  
    If ReqTable.Fields("TABLE_TYPE") = "TABLE" Then  
        Fich.WriteLine "Nom : " & ReqTable!table_name & " type: " &  
ReqTable!table_type & " description : " & ReqTable!Description  
        NomTab = ReqTable.Fields("TABLE_NAME")  
        Set ReqChamp = cnn1.OpenSchema(adSchemaColumns, Array(Empty,  
Empty, NomTab, Empty))  
        Fich.Write vbTab  
        For compteur = 0 To ReqChamp.Fields.Count - 1  
            Fich.Write ReqChamp.Fields(compteur).Name & vbTab  
        Next compteur  
        Fich.WriteLine 1  
        Fich.WriteLine ReqChamp.GetString(adClipString, -1, vbTab,  
vbCrLf)  
        End If  
        ReqTable.MoveNext  
    Loop  
Fich.Close  
End Sub
```

Ce code récupère le schéma des tables dans le recordset ReqTable, et pour chaque table extrait le schéma des colonnes. Observons la ligne suivante :

```
Set ReqChamp = cnn1.OpenSchema(adSchemaColumns, Array(Empty, Empty, NomTab,  
Empty))
```

Si nous regardons l'aide nous trouvons dans le tableau :

QueryType	Criteria
adSchemaColumns	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME

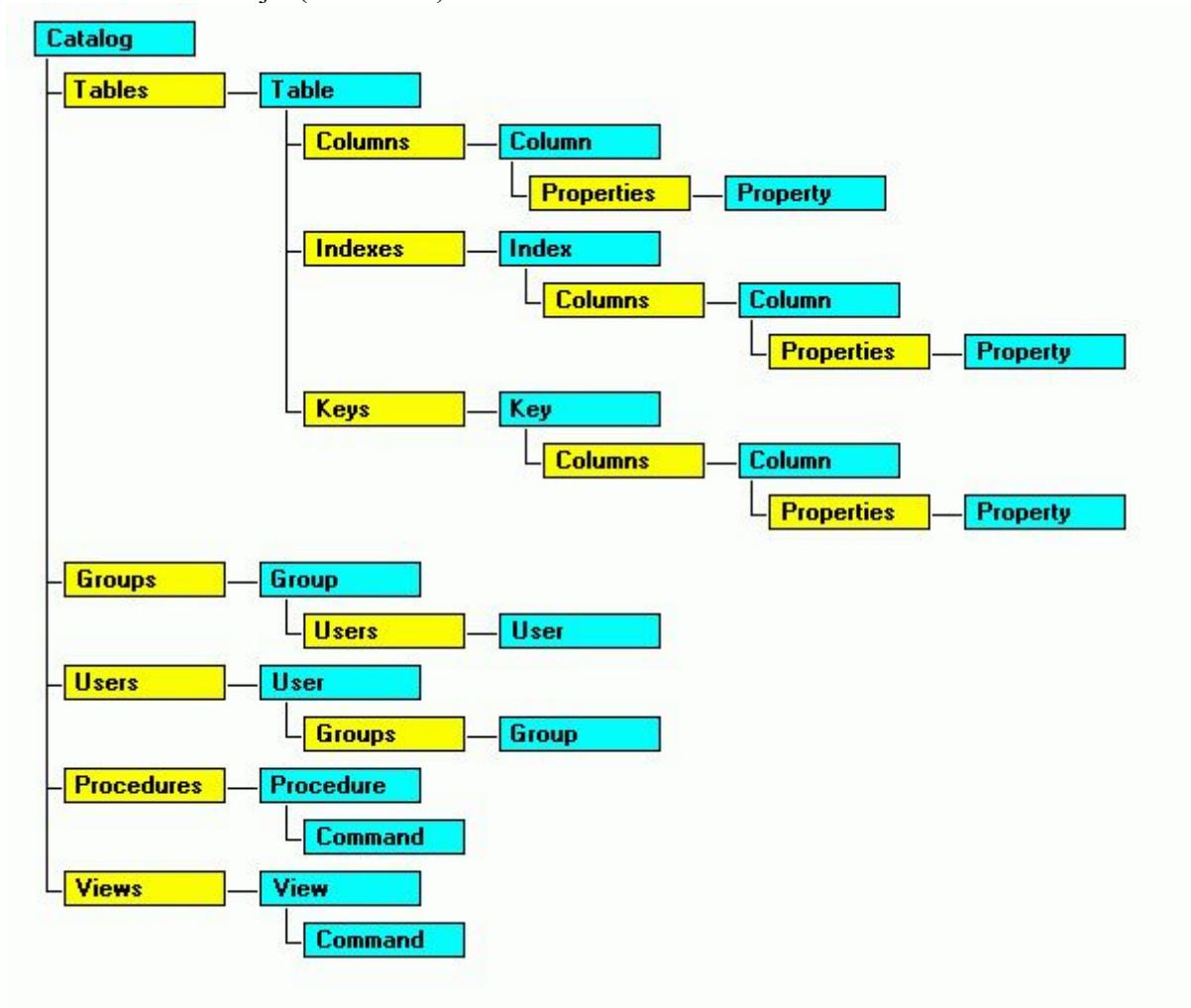
C'est donc pour obtenir la liste des champs d'une table que j'utilise Array(Empty, Empty, NomTab, Empty). Notez bien que si dans la ligne je remplace "NomTab" par "ReqTable.Fields("TABLE\_NAME")" j'obtiens une erreur du fournisseur.

Les constantes QueryType n'étant pas toujours très claires, et de manière générale pour comprendre les notions de schéma et de catalogue ➔ [http://sqlpro.developpez.com/SQL\\_AZ\\_7.html](http://sqlpro.developpez.com/SQL_AZ_7.html)

 De nombreuses valeurs de QueryType pourraient déclencher une erreur. En effet, tout ce qui est lié à un utilisateur demanderait une connexion intégrant le fichier de sécurité (.mdw) pour peu que celui-ci ne soit pas celui enregistré dans la clé de registre. Nous verrons ce point plus en détail lors de l'étude de la sécurité.

# Modèle objet

Le modèle objet (ADOX 2.6) suivi est le suivant :



## Notions Fondamentales

### ADOX & Access

Dans ce modèle, il y a une certaine redondance. Dans le cas d'Access, une requête non paramétrée renvoyant des enregistrements est considérée comme une table de type "VIEW" (vues). Une telle requête apparaîtra donc comme membre de la collection "Table" (un peu différente des autres tables) et comme membres de la collection "Views" mais pas comme membre de la collection procédure.

Attention, dans Access il faut considérer une procédure comme une requête paramétrée ou une requête ne renvoyant pas d'enregistrement (Requête action par exemple) ; n'attendez pas de récupérer des procédures VBA dans cette collection.

## Propriétaire

Cette notion de propriété est très importante. En effet seul le propriétaire d'un objet peut faire certaines modifications sur celui-ci et attribuer les droits sur ces objets. Par défaut, le propriétaire d'un objet est son créateur. Cela implique deux choses :

Ne pas laisser aux utilisateurs le droit de créer des objets dans vos bases sans un contrôle car vous n'auriez pas de droit de modifications sur ces objets sans en reprendre la propriété.

Gérer strictement les droits sur la base.

Nous y reviendrons plus en détail lorsque nous aborderons la sécurité.

## ParentCatalog

Lorsque l'on crée des objets, ils ne sont reliés à aucun catalogue. Dès lors, comme ils ne connaissent pas leurs fournisseurs, il n'est pas possible de valoriser leurs propriétés. On définit donc la propriété ParentCatalog d'un objet dès que l'on souhaite valoriser ses propriétés. Ceci pourtant dissimule un problème d'un fort beau gabarit ma foi. Outre le fournisseur, le catalogue transmet aussi ses droits de propriétaire. Si on n'y prend pas garde, il se peut que l'objet se retrouve ayant des droits non souhaités.

## L'objet Catalog

C'est l'objet racine du modèle.

### ActiveConnection (Prop. S)

C'est la seule propriété de l'objet Catalog, qui lui permet de définir le fournisseur de données, la base de données et éventuellement les paramètres de sécurité. Cette connexion est stricto sensu une connexion ADO, on peut donc parfaitement utiliser une connexion existante. Attention toutefois, ce sont les paramètres de la connexion qui définissent en partie les possibilités du fournisseur, il est donc vivement recommandé de créer sa connexion pour la gestion de la sécurité. Les deux codes suivants permettent de définir la connexion :

```
Dim cnn1 As New ADO.DB.Connection, Catalogue As ADOX.  
Set Catalogue = New ADOX.Catalog  
With cnn1  
    .Provider = "Microsoft.Jet.OLEDB.4.0;"  
    .Properties("Jet OLEDB:System database") =  
"D:\User\jmarc\tutorial\ADOX\system.mdw"  
    .Open "Data Source=D:\User\jmarc\tutorial\ADOX\baseheb.mdb ;User  
Id=Admin; Password="  
End With  
Catalogue.ActiveConnection = cnn1
```

Ou

```
Catalogue.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & "Data  
Source=:\User\jmarc\tutorial\ADOX\baseheb.mdb;Jet OLEDB:System database=" &  
"D:\User\jmarc\tutorial\ADOX\system.mdw;User Id=Admin; Password="
```

### Create (Mth.)

Permet la création d'une nouvelle base de données selon les paramètres de la chaîne de connexion passée en paramètre. Attention, le nom de la base de données ne doit pas être une base existante. Le paramètre à passer à la méthode doit être une chaîne de connexion valide.

### GetObjectOwner & SetObjectOwner (Mth.)

De la forme

Catalog.SetObjectOwner [ObjectName](#), [ObjectType](#), [OwnerName](#) [[ObjectTypeId](#)]

Owner = Catalog.GetObjectOwner([ObjectName](#), [ObjectType](#) [[ObjectTypeId](#)])

Permet de renvoyer ou de définir le propriétaire d'un objet de la base, celui-ci pouvant être la base de données, une table, un champ.....

Ne fonctionne qu'avec un fichier de sécurité défini.

### Les collections

L'objet Catalog renvoie aussi cinq collections qui représentent la structure de la base. Nous allons maintenant étudier ces objets

## Collections de l'objet Catalog

Dans ADOX, les collections ont normalement une propriété Count, et quatre méthodes. Attention le premier index d'une collection est zéro.

### Append

Ajoute un objet à la collection. La méthode Append est propre à chaque collection, aussi allons nous la voir en détail plus loin. Notons toutefois qu'un objet créé est librement manipulable, tant qu'il n'a pas été ajouté à la collection correspondante. Après l'ajout, un certain nombre de propriétés passe en lecture seule.

### Item

Renvoie un élément d'une collection par son nom ou son index.

### Delete

Retire un élément d'une collection. Il faut faire très attention à la suppression de certains éléments qui peut mettre en danger l'intégrité des données voire endommager la base

### Refresh

Met à jour la collection.

## Collection tables

Représente l'ensemble des tables, au sens large du terme, qui compose la base de données. Ces tables peuvent être rangées dans les familles suivantes :

- ✓ Table système
- ✓ Table Temporaire
- ✓ Table de données
- ✓ Vues

En générale on ne travaille que sur les tables de données à l'aide de l'objet Table.

### Append

De la forme `Cat.Tables.Append NomTable`. Lors de la création tous les objets appartenant à la table doivent être créés avant l'ajout de celle-ci à la collection

## Collection Procedures

Représente l'ensemble des requêtes définies dans la base de données à l'exclusion des requêtes non paramétrées renvoyant des enregistrements.

### Append

De la forme `Cat.Procedures.Append NomRequete, Command`. Command représente un objet command qui représente la requête.

## Collection Views

Représente l'ensemble des requêtes non paramétrées renvoyant des enregistrements. On les retrouve d'une certaine façon dans la collection Tables avec Access.

### Append

De la forme `Cat.Views.Append NomRequete, Command`. Command représente un objet command qui représente la requête.

## Collection Groups

Représente l'ensemble des groupes définis dans la base. A ne pas confondre avec la collection Groups de l'objet User qui est le groupe auquel appartient un utilisateur.

### Append

De la forme `Cat.Groups.Append NomGroupe`. Ceci représente l'ajout d'un groupe à la base de données. Les permissions doivent être définies lors de l'ajout du groupe à la collection. Le groupe doit être ajouté à la collection avant la création d'un utilisateur du groupe.

## Collection Users

Représente l'ensemble des utilisateurs définis dans la base. A ne pas confondre avec la collection Users de l'objet Group qui est l'ensemble des utilisateurs définis dans un groupe.

### Append

De la forme `Cat.Users.Append NomUtilisateur [, MotDePasse]`. Ceci représente l'ajout d'un utilisateur à la base de données.

## L'objet Table

Cet objet représente une table de la base de données. Les tables de type "VIEW" seront vues dans le chapitre du même nom. Nous allons donc regarder les tables Standard.

Une table contient des champs (colonnes), des index, des clés et des propriétés propres au fournisseur.

Seules deux propriétés intéressantes sont statiques : « Name » qui représente le nom de la table et est obligatoire, et « type » qui porte bien son nom mais qui est en lecture seule. Ceci fait qu'il n'y a pas de questions existentielles à se poser, lorsqu'on crée une table avec ADOX, elle est forcément standard.

La table n'existe réellement dans la base de données que lorsque

- Son catalogue est défini
- Elle est ajoutée à la collection Tables.

Il convient donc de créer entièrement la table avant de l'ajouter à la collection.

## Collection Properties

Cette collection n'est utilisée que dans le cas des tables liées. N'essayez pas de créer des tables liées en partant du catalogue de votre base, bien que techniquement possible cela n'est pas conseillé. Le code suivant crée une table liée

```
Sub CreateAttachedJetTable()  
Dim Catalogue As ADOX.Catalog, MaTable As ADOX.Table  
Set Catalogue = New ADOX.Catalog  
Catalogue.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" &  
"Data Source=D:\ADOX\baseheb.mdb;Jet OLEDB:System database=" & "D:\  
ADOX\system.mdw;User Id=Admin; Password="  
Set MaTable = New ADOX.Table  
MaTable.Name = "auteurs"  
Set MaTable.ParentCatalog = Catalogue  
MaTable.Properties("Jet OLEDB:Create Link") = True  
MaTable.Properties("Jet OLEDB:Link Datasource") = "D:\adox\Biblio.mdb"  
MaTable.Properties("Jet OLEDB:Link Provider String") = ";Pwd=password"  
MaTable.Properties("Jet OLEDB:Remote Table Name") = "auteurs"  
Catalogue.Tables.Append MaTable  
Set Catalogue = Nothing  
End Sub
```

Lorsque j'écris que c'est déconseillé, ce n'est pas dû à la complexité du code mais pour les deux raisons suivantes :

- Il faut s'assurer que le fournisseur utilisé pour faire la liaison sera présent sur le poste qui exécute.
- Il faut bien savoir si les paramètres de sécurité de liaison seront ou ne seront pas stockés de façon permanente dans la base.

Regardons ce code. Après avoir défini un catalogue nous créons un objet Table. Dans cet exemple, le nom est le même que le vrai nom de la table dans l'autre base, mais cela n'a aucune importance. Ensuite je lui attribue un catalogue afin de pouvoir valoriser ses propriétés dynamiques. Notez bien que j'utilise l'objet Catalog de ma base car je crée une table liée dans ma base.

Je valorise ensuite mes propriétés dynamiques, notons que je ne mets pas de fournisseur dans "Jet OLEDB:Link Provider String" car je continue d'utiliser Jet mais je pourrais tout à fait changer le fournisseur. Enfin j'ajoute ma table à la collection. A partir de cet ajout, la table liée est créée mais un certain nombre de ses propriétés sont passées en lecture seule.

## Collection Columns



Il existe des collections/objets Column pour l'objet Table, Index et Key. Attention de ne pas les confondre.

La collection Columns regroupe les champs de la table. Elle possède les propriétés/méthodes d'une collection normale.

**Append (méth.)** : De la forme

*Columns.Append Column [, Type] [, DefinedSize]*

Où Column est l'objet colonne à ajouter.

### Objet Column

Représente dans ce cas un champ de la table. A ne pas confondre avec l'objet Column de l'index. Cet objet n'a pas de méthode.

#### Propriétés

##### **Attributes**

Défini si la colonne est de longueur fixe et si elle accepte les valeurs NULL.

##### **DefinedSize**

Défini la taille maximum du champ

##### **Name**

Nom de la table. Obligatoire. L'unicité d'un nom dans une collection n'est pas obligatoire, mais dans une même base, il peut pas y avoir deux tables portant le même nom.

##### **NumericScale**

A ne pas confondre avec la propriété Précision. Donne l'échelle (nombre de chiffre après la virgule)d'un champ dont le type est adNumeric ou adDecimal.

##### **ParentCatalog**

Qu'on ne présente plus!

##### **Precision**

Défini la précision (nombre maximal de chiffre pour représenter la valeur)d'un champ numérique.

##### **Type**

Défini le type du champ. Il existe beaucoup de types définis (39) mais le tableau suivant vous donnera les principaux types Access.

Type	Valeur	Commentaire
adSmallInt	2	Type Entier court
adInteger	3	Type Entier long (Access)
adSingle	4	Valeur décimale à simple précision
adDouble	5	Valeur décimale à double précision
adCurrency	6	Type monétaire. Utilise normalement 4 chiffres après la virgule
adDate	7	Stocké comme un double. La partie entière étant le nombre de jours depuis le 30/12/1899
adBoolean	11	Booléen
adUnsignedTinyInt	17	Numérique de type Octet
adGUID	72	Par exemple numéro de réplication
adVarChar	202	Chaîne de caractères (type Text Access)
adLongVarChar	203	Champs mémo (Access) et lien hypertexte
adLongVarBinary	205	Type OLE Object Access

### Collection Properties

Les propriétés dynamiques de l'objet Column intéressantes sont :

**Autoincrement** : Permet de créer les colonnes à numérotation automatique.

**Default** : Définit la valeur par défaut d'un champ.

**Nullable** : Définit si la colonne accepte des valeurs NULL.

**Fixed Length** : Définit si la colonne est de longueur fixe.

**Seed** : Détermine la prochaine valeur qui sera fourni par un champ de type numéro Automatique.

**Increment** : Définit le pas d'incrément d'un champ NuméroAuto.

**Jet OLEDB:Column Validation Text** : Définit le message d'alerte si une règle de validation n'est pas respectée.

**Jet OLEDB:Column Validation Rule** : Définit une règle de validation pour un champ.

**Jet OLEDB:Allow Zero Length** : Définit si les chaînes vides sont autorisées.

### Exemple

Dans l'exemple suivant, nous allons créer une table contenant trois champs, Identification de type NuméroAuto, Nom de type Texte, Ntel de type numérique formaté.

```
Dim Catalogue As ADOX.Catalog, MaTable As ADOX.Table
Dim strConn As String, compteur As Long, MaCol As ADOX.Column

Set Catalogue = New ADOX.Catalog
strConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" &
"D:\User\jmarc\tutorial\ADOX\Test\NouvBase1.mdb"
Catalogue.Create strConn
Set MaTable = New ADOX.Table
MaTable.Name = "PremTable"
For compteur = 1 To 3
    Set MaCol = New ADOX.Column
    With MaCol
        .DefinedSize = Choose(compteur, 0, 50, 14)
        .Name = Choose(compteur, "Id", "Nom", "Tel")
        .Type = Choose(compteur, adInteger, adVarChar, adVarChar)
        .Attributes = Choose(compteur, 0, adColFixed, adColFixed +
adColNullable)
    End With
    If compteur = 1 Then
        Set .ParentCatalog = Catalogue
        MaCol.Properties("Autoincrement") = True
        MaCol.Properties("Seed") = CLng(1)
        MaCol.Properties("Increment") = CLng(1)
    End If
    End With
    MaTable.Columns.Append MaCol
    Set MaCol=Nothing
Next compteur
Catalogue.Tables.Append MaTable
End Sub
```

Je commence donc par créer la base de données. Pour cela j'utilise une chaîne de connexion qui est obligatoire pour pouvoir utiliser la méthode Create de catalogue. Dans cet exemple, je ne définis pas de fichier de sécurité. Ensuite je crée mon objet Table. Comme je n'utilise pas sa collection Properties, je ne valorise pas sa propriété ParentCatalog.

Dans la boucle, je crée mes colonnes. J'utilise la fonction Choose afin d'écrire de façon plus concise. Pour chaque colonne je définis une taille, un nom, un type et éventuellement ses attributs. Notons que comme la colonne n'est pas encore ajoutée à la collection Columns, elle n'existe pas encore. Dès lors, l'ordre de valorisation des propriétés n'a aucune importance. Dans le cas de la colonne "Id" je vais créer un champ NuméroAuto. Pour cela, je dois valoriser des propriétés dynamiques, donc je dois définir la propriété ParentCatalog. Enfin j'ajoute la colonne à la collection Columns puis la table à la collection Tables.

Notez aussi que tous ce code revient au même, à l'exception de la création de la base que la commande DDL

```
"CREATE TABLE PremTable(Id COUNTER, Nom TEXT 50 NOT NULL, Tel TEXT 14)"
```

## **Collection Indexes**

Représente les index de la table. Faites bien attention à ne pas confondre index et clé. Un index sert à augmenter l'efficacité du SGBDR lors des recherches, de la mise en relation etc....

Normalement devrait toujours être indexés

- ❖ Les colonnes composantes d'une clé
- ❖ Les colonnes ayant des contraintes de validité ou d'unicité
- ❖ Les colonnes étant la cible de nombreuses recherches.

Les propriétés/méthodes de cette collection sont celles des collections standard si ce n'est :

### **Append**

De la forme : *Indexes.Append Index* [, *Columns*]

Où *Columns* renvoie le ou les noms des colonnes devant être indexées. En général, on définit l'objet *Columns* avant son ajout à la collection

### **Objet Index**

Cet objet n'a pas de méthodes.

Attention, on ne peut utiliser un objet *Column* avec cet objet que si la colonne appartient déjà à la collection *Columns* de l'objet *Table*.

#### **Clustered**

Indique si l'index est regroupé. Un index regroupé signifie qu'il existe un ordre physique pour les données. C'est en général l'index de la clé primaire.

#### **IndexNulls**

Définit le comportement de l'index vis à vis des colonnes ayant une valeur NULL.

#### **PrimaryKey**

Précise si l'index représente la clé primaire de la table.

NB : C'est là que je vous disais de faire attention. Une clé primaire (ou non d'ailleurs) est définie par la collection *Keys*. Néanmoins on définit en général aussi un index sur la colonne de clé primaire. En mettant à vrai cette propriété, la création de l'index implique la création de la clé. Donc si vous créez une clé primaire de façon implicite n'essayez pas de la recréer après dans la collection *Keys* et vice versa.

#### **Unique**

Définit la contrainte d'unicité.

#### **Collection Properties**

Comme toujours, la propriété *ParentCatalog* de l'index doit être valorisé avant l'appel d'un membre de cette collection. La plupart de ces propriétés étant en lecture seule ou redondante avec les propriétés statiques de l'objet, nous n'irons pas plus avant dans notre exploration.

#### **Collection Columns et objet Column**

Définit les colonnes sur lesquelles porte l'index. N'oubliez pas qu'un index porte sur un ou plusieurs champs mais qu'il appartient à l'objet *Table*.

L'objet *Column* doit impérativement exister dans la table avant de pouvoir être ajouté à la collection *Columns* de l'index.

Les colonnes des index utilisent une propriété supplémentaire

#### **SortOrder**

Définit l'ordre de tri d'une colonne d'index.

### Exemple

Dans mon exemple précédent je vais rajouter un index sur la colonne Nom, avec un tri croissant.

```
Set MonIndex = New ADOX.Index
With MonIndex
    .IndexNulls = adIndexNullsDisallow
    .Name = "ind1"
    .Columns.Append "Nom"
    .Columns("Nom").SortOrder = adSortAscending
End With
MaTable.Indexes.Append MonIndex
```

Ce code se situe après l'ajout des colonnes à l'objet Tables mais avant l'ajout de l'objet Table. Comme précédemment, je manipule mon objet avant son ajout à la collection.

✿ Seules les colonnes indexées supportent l'emploi de la méthode de recherche Seek du recordset ADO. Celle ci est beaucoup plus performante que la méthode Find.

## **Collection Keys**

Nous allons aborder maintenant un des passages un peu "sensibles" de la programmation ADOX, c'est à dire les clés. Les clés sont le fondement de l'intégrité référentielle il convient de bien connaître le sujet avant de se lancer dans leur programmation. Normalement, vous avez travaillé sur le modèle de données avant de construire votre base donc il suffira de suivre celui-ci.

### Quelques notions

Les clés sont parfois appelées, improprement d'ailleurs, contraintes. Ceci vient du fait qu'en SQL, Primary Key est une contrainte. Pourtant si les clés peuvent être des contraintes, toutes les contraintes ne sont pas des clés.

#### Clé primaire

Cette clé devrait être composée d'une seule colonne mais ce n'est pas une obligation. Elle sert à identifier chaque enregistrement de manière unique. La clé primaire n'accepte pas les valeurs NULL. Beaucoup de gens utilisent un champ à numérotation automatique pour définir la clé primaire, cela n'est en rien obligatoire. Une table n'a qu'une clé primaire.

#### Clé étrangère

Champ de même type/sous-type de données que la clé primaire, qu'elle référence dans la table enfant. Une table peut posséder plusieurs clés étrangères.

#### Intégrité référentielle

Mécanisme de vérification qui s'assure que, pour chaque valeur de clé étrangère, il y a une valeur de clé primaire lui correspondant. Ce mécanisme se déclenche lors de l'ajout d'une nouvelle valeur de clé étrangère, lors de la suppression d'une clé primaire ou lors de la modification de l'une quelconque des deux clés.

#### Opération en cascade

Mode opératoire dépendant de l'intégrité référentielle. Celui-ci détermine éventuellement le comportement du SGBDR en cas de suppression/modification d'un enregistrement parent.

## **Méthode Append**

De la forme *Keys.Append Key [, KeyType] [, Column] [, RelatedTable]*

Nous examinerons plus tard le détail des paramètres (dans les propriétés de l'objet Key).

## **Objet Key**

Représente une clé primaire, étrangère ou unique.

### **DeleteRule**

Définit les règles à appliquer lors de la suppression d'une valeur de la clé primaire. On distingue 4 cas :

#### **AdRINone**

Aucune action (équivalent du mode de gestion SQL ON DELETE NO ACTION). Cette valeur interdit toute suppression d'un enregistrement tant qu'il existe un enregistrement dépendant dans la base.

#### **AdRICascade**

Modifications en cascade (SQL ON DELETE CASCADE). Attention à l'utilisation de cette valeur. Elle supprime tous les enregistrements liés lors de la suppression de l'enregistrement père. Ceci peut être très coûteux pour le SGBD en termes de performances et doit être limité au sein d'une transaction

#### **AdRISetNull & adRISetDefault**

Attribue la valeur nulle ou la valeur par défaut à la clé étrangère. Ceci se justifie rarement sauf pour gérer des traitements par lots à posteriori.

### **Name**

Tel que son nom l'indique

### **RelatedTable**

Si la clé est une clé étrangère, définit le nom de la table contenant la clé primaire correspondante.

### **Type**

Détermine s'il s'agit d'une clé primaire, étrangère ou unique. Une clé unique est une clé sans doublons.

### **UpdateRule**

Equivalent pour les modifications à DeleteRule pour les mises à jour.



Attention de nombreux SGBDR n'acceptent pas toutes les règles de modification/suppression. Je vous rappelle aussi que ces règles s'appliquent sur la clé étrangère ; les définir pour la clé primaire ne sert à rien.

### **Collection Columns et objet Column**

Définit la ou les colonne(s) appartenant à la clé. Celles-ci doivent déjà appartenir à la collection Columns de l'objet Table. L'objet Column de l'objet Key utilise la propriété **RelatedColumn**. Celle-ci définit la colonne en relation dans la table définie par RelatedTable.

## Exemple

Nous allons voir maintenant un exemple de création de clé.

```
Private Sub Command7_Click()  
Dim Catalogue As ADOX.Catalog, MaTable As ADOX.Table, compteur As Long  
Dim MaCol As ADOX.Column, MaCle As ADOX.Key, MonIndex As ADOX.Index  
Set Catalogue = New ADOX.Catalog  
.....'ici se trouve le code précédemment vu  
Set MaCle = New ADOX.Key  
With MaCle  
    .Type = adKeyPrimary  
    .Name = "ClePrim"  
    .Columns.Append "Id"  
End With  
MaTable.Keys.Append MaCle  
Set MaCle = Nothing  
Catalogue.Tables.Append MaTable  
Set MaTable = New ADOX.Table  
MaTable.Name = "DeuxTable"  
For compteur = 1 To 3  
    Set MaCol = New ADOX.Column  
    With MaCol  
        .DefinedSize = Choose(compteur, 0, 255, 0)  
        .Name = Choose(compteur, "NumCle", "Local", "Responsable")  
        .Type = Choose(compteur, adInteger, adVarChar, adInteger)  
        Set .ParentCatalog = Catalogue  
        .Attributes = Choose(compteur, 0, adColFixed, 0)  
    End With  
    MaTable.Columns.Append MaCol  
    Set MaCol = Nothing  
Next compteur  
For compteur = 1 To 2  
    Set MonIndex = New ADOX.Index  
    With MonIndex  
        .PrimaryKey = Choose(compteur, True, False)  
        .Unique = Choose(compteur, True, False)  
        .IndexNulls = adIndexNullsDisallow  
        .Name = Choose(compteur, "ind21", "Ind22")  
        .Columns.Append Choose(compteur, "NumCle", "Responsable")  
        .Columns(Choose(compteur, "NumCle", "Responsable")).SortOrder =  
adSortAscending  
    End With  
    MaTable.Indexes.Append MonIndex  
    Set MonIndex = Nothing  
Next compteur  
Set MaCle = Nothing  
Set MaCle = New ADOX.Key  
With MaCle  
    .DeleteRule = adRINone  
    .UpdateRule = adRINone  
    .Type = adKeyForeign  
    .Name = "CleEtr1"  
    .RelatedTable = "PremTable"  
    .Columns.Append "Responsable"  
    .Columns("Responsable").RelatedColumn = "Id"  
End With  
MaTable.Keys.Append MaCle  
Set MaCle = Nothing  
Catalogue.Tables.Append MaTable  
End Sub
```

Dans ma première table je crée une clé primaire de façon explicite. Je lui donne des règles NO ACTION. Je crée ensuite une deuxième table contenant trois champs. Le champ "NumCle" quoique n'étant pas de type NuméroAuto, doit être unique et non NULL. Le champ "Responsable" est la clé étrangère et contient la valeur du champ "Id" correspondant. Ces deux colonnes sont indexées. En jouant sur la propriété PrimaryKey de l'index, je crée implicitement la clé primaire en créant l'index. Notons que cette clé portera le même nom que l'index. Je crée ensuite ma clé étrangère et me voilà avec deux tables mises en relation.

## **Conclusion sur les tables**

Comme nous l'avons vu au cours de ce chapitre, le schéma de construction est assez simple. Voici les quelques règles à se rappeler :

- On crée les tables les unes après les autres en commençant toujours par les tables parents (celles qui n'ont pas de clés étrangères)
- On ajoute un objet à sa collection lorsqu'on l'a entièrement défini.
- On crée les champs de la table avant de définir les index et les clés.
- On indexe toujours les champs intervenants dans les jointures et/ou cible de recherche.
- La définition des clés lors de la création de la base augmente la sécurité des données.

## **L'objet Procedure**

Cet objet peut représenter plusieurs choses selon les cas.

- Une requête Action
- Une requête paramétrée
- Une procédure stockée (n'existe pas dans Access)

En soi cet objet est très simple à utiliser puisqu'il ne s'agit que d'un objet Command ayant un nom dans la collection Procedures. Comme nous avons vu plus haut l'objet Command suffisamment en détail, nous allons nous pencher dans ce chapitre sur l'ajout ou la modification de l'objet Procedure.

## **Création d'un objet procédure**

Elle se fait en générale en utilisant la méthode Append de la collection Procedures de la forme *Catalog.Procedures.Append Name, objCommand*

Où *objCommand* est un objet Command. Tout se fait donc dans la définition de l'objet Command.

Dans le cas d'une requête action, la création est facile puisque le texte de la commande est le code SQL. Mais dès lors qu'il y a utilisation de paramètres il faut faire attention.

## **Pas d'objet Parameter**

Nous avons vu que pour l'appel des procédures stockées ou des requêtes paramétrées, nous utilisons les objets Parameters de la commande. Lors de la création cela n'est pas possible. La propriété CommandText de l'objet Command Doit contenir l'ensemble de la requête / procédure telle qu'elle est effectivement écrite dans le SGBD.

A part cela, la création d'un objet Procedure ne pose aucun problème.

### Exemple

Dans la suite de mon exemple je peux ajouter

```
Set MaCommand = New ADODB.Command
MaCommand.CommandText = "PARAMETERS [QuelNom] TEXT(50);SELECT * FROM
PremTable WHERE Nom = [QuelNom]"
Catalogue.Procedures.Append "ParNom", MaCommand
```

Ce code ajoute une requête paramétrée nommée "ParNom" attendant le paramètre "QuelNom" de type texte.

Il est tout à fait possible de créer une requête action paramétrée. Par exemple

```
PARAMETERS DateCib DateTime, BancCib Text(255);DELETE * FROM tblAnomalie
WHERE NumBanc=[BancCib] AND DatDimanche>=[DateCib];
```

Est une requête valide.

 **Attention** : Pour les paramètres, il faut bien préciser la taille dans le corps de la procédure.

En effet, si j'avais écrit PARAMETERS [QuelNom] TEXT, sans préciser une taille inférieure ou égale à 255 j'aurais créé un paramètre de type mémo au lieu de texte. (**adLongVarChar au lieu de adVarChar**). Ce genre d'erreur engendre des bugs très difficiles voir impossibles à retrouver pour les utilisateurs de vos bases de données.

## **Modification d'un objet Procedure**

La modification de l'objet Procédure passe par la modification de l'objet Command sous-jacent. Donc en générale on affecte à un objet Command l'objet Command de l'objet Procedure, on modifie le texte et on fait l'opération inverse. Par exemple :

```
Set MaCommand = Catalogue.Procedures("ParNom").Command
MaCommand.CommandText = "PARAMETERS [QuelLocal] TEXT(255);SELECT * FROM
DeuxTable WHERE Local = [QuelLocal]"
Set Catalogue.Procedures("ParNom").Command = MaCommand
```

« Mais alors, pourquoi ne pas écrire ? »

```
Catalogue.Procedures("ParNom").Command.CommandText= "PARAMETERS [QuelLocal]
TEXT(255);SELECT * FROM DeuxTable WHERE Local = [QuelLocal]"
```

La raison est la suivante :

Intrinsèquement, un objet Command est volatile, c'est à dire qu'il n'est pas lié à la base de données (à contrario des objets DAO QueryDefs). Si nous utilisons le code ci-dessus, la procédure se comporterait comme attendu au cours de la même session de programme mais les modifications ne seraient pas enregistrées dans la base de données et ces modifications seraient perdues. C'est pourquoi il convient de réaffecter explicitement un objet Command à l'objet Procedure.

## **L'objet View**

La création et la modification d'un objet View se gère de la même manière que pour les objets Procedure. C'est pourquoi nous ne nous rééditerons pas ces notions ici. Il faut cependant savoir que la seule vraie différence existant entre l'Objet Procédure et l'objet View sont que ce dernier n'accepte que des requêtes Select sans paramètre.

## Conclusion sur les objets View & Procedure

Ces deux objets ne présentent aucune difficulté majeure à programmer si ce n'est de faire attention à toujours affecter les objets Command de façon explicite.

Dans le cas des requêtes renvoyant des enregistrements (View ou Procedure) on peut directement valoriser un recordset avec.

Ne perdez pas de temps à paramétrer l'objet Command lors de la création de requête ou de procédure, seule la propriété CommandText compte.

Pour se finir voilà un code qui permet de récupérer l'ensemble des requêtes / procédures de la base de données sous leurs formes SQL.

```
Private Sub Command1_Click()  
Dim cnn1 As ADODB.Connection  
Dim Catalogue As ADOX.Catalog, fso As New FileSystemObject  
Dim Fich As TextStream, MaCmd As ADODB.Command  
Dim MaProc As ADOX.Procedure, MaVue As ADOX.View  
  
Set Fich = fso.CreateTextFile("d:\recupTexte.txt", True)  
Set Catalogue = New ADOX.Catalog  
Set cnn1 = New ADODB.Connection  
With cnn1  
    .Provider = "Microsoft.Jet.OLEDB.4.0;"  
    .ConnectionTimeout = 30  
    .CursorLocation = adUseClient  
    .IsolationLevel = adXactChaos  
    .Mode = adModeShareExclusive  
    .Properties("Jet OLEDB:System database") =  
"D:\User\jmarc\tutorial\ADOX\system.mdw"  
    .Open "Data Source=D:\User\jmarc\tutorial\ADOX\baseheb.mdb ;User  
Id=Admin; Password="
```

```
End With  
Catalogue.ActiveConnection = cnn1  
For Each MaProc In Catalogue.Procedures  
    Fich.WriteLine "NOUVELLE PROCEDURE"  
    Fich.WriteLine "Nom = " & MaProc.Name  
    Set MaCmd = MaProc.Command  
    Fich.WriteLine MaCmd.CommandText  
Next  
For Each MaVue In Catalogue.Views  
    Fich.WriteLine "NOUVELLE VUE"  
    Fich.WriteLine "Nom = " & MaVue.Name  
    Set MaCmd = MaVue.Command  
    Fich.WriteLine MaCmd.CommandText  
Next  
Fich.Close  
End Sub
```

Voilà vous savez maintenant créer par le code une base de données, nous allons attaquer une partie beaucoup plus complexe, mais néanmoins essentielle, la gestion des utilisateurs.

## Gestion des utilisateurs

La gestion des utilisateurs comprend en fait de façon imbriquée deux thèmes que sont les droits d'accès et la propriété. Quasiment tous les SGBD disposent d'un système limitant l'accès aux données ainsi qu'à la structure de la base. Avec ADOX on retrouve ces concepts dans les objets User et Group.

### Cas particulier d'Access

Les concepts que nous allons voir plus loin fonctionnent pour de nombreux SGBD mais Access gère la sécurité utilisateur par l'utilisation d'un fichier externe. Ceci présente des avantages et des inconvénients mais je ne vais pas démarrer une polémique ici.

Si j'aborde ce point c'est pour une particularité d'Access. Pour garantir l'unicité de chaque utilisateur ou groupe, le fichier de sécurité Access utilise un identifiant Unique nommé SID qui est généré à l'aide du nom de l'utilisateur (ou du groupe) et d'une chaîne nommée PID. Un même nom avec un même PID génère toujours le même SID ce qui permet de reconstruire le fichier de sécurité si besoin est.

Si cette possibilité existe avec DAO, ADO ne permet pas de fournir un PID explicite ce qui rend le fichier impossible à reconstruire en cas de problème. Certes des sauvegardes fréquentes doivent pouvoir éviter ce problème mais c'était une perte de souplesse importante.

De même, normalement un utilisateur hérite implicitement des droits d'accès du groupe, mais cela n'est pas le cas avec ADO 2.7 et inférieur. Il faut alors coder explicitement les droits du groupe et de l'utilisateur.

Comme Access gère sa sécurité par un fichier externe, il est fortement conseillé de spécifier ce fichier dans la connexion du catalogue avec la Property ("Jet OLEDB:System database") et éventuellement la création d'un nouveau fichier avec la Property (Jet OLEDB:Create System Database)



Si vous spécifiez un fichier de sécurité incorrecte, vous n'aurez pas d'erreur lors de l'ouverture du catalogue. L'erreur surviendra lors de l'appel d'un objet lié à la sécurité. De plus vous obtiendrez une erreur 3251 " L'opération demandée par l'application n'est pas prise en charge par le fournisseur." ce qui ne vous aidera pas beaucoup à voir l'erreur.

### Propriétés et droits

Voilà deux notions importantes, corrélatives et pourtant différentes dans leur conception.

#### Propriétaire

Chaque objet a **toujours** un propriétaire. A l'origine, c'est le créateur de l'objet qui est le propriétaire.

Le propriétaire d'un objet garde le droit d'accès total (modification, suppression, droits) sur l'objet. Le propriétaire d'un objet peut être un utilisateur ou un groupe. La base de données est un objet un peu particulier puisque seul un utilisateur (et non un groupe) peut en être propriétaire.

Le problème de la gestion de la propriété est important à gérer si les utilisateurs ont le droit de créer des objets. Si vous souhaitez qu'ils gardent tous les droits sur les objets qu'ils ont créés, il n'y a rien à faire, mais sinon vous devez gérer le transfert de la propriété.

Faites toujours très attention à cette notion de propriétaire, elle peut permettre des accès que vous ne souhaiteriez pas si vous n'y prenez pas garde.

## **Administrateur**

Habituellement, l'administrateur ou les membres du groupe administrateur sont les seuls à avoir tous les droits sur une base de données, y compris celui de modifier les droits d'accès des autres. De part ce fait, on essaye toujours de limiter le nombre d'administrateurs. Pour ce faire, il faut donc mettre au point une stratégie de droits pour que chaque groupe ait les droits nécessaires et suffisants à un bon fonctionnement. L'avantage dans ce cas d'une approche par programme est la possibilité d'utiliser une connexion ayant les droits administrateurs pour permettre aux utilisateurs de faire des actions (prévues) que leurs droits ne leur permettraient pas de faire normalement.

### **Access**

Même si cela est transparent, la sécurité d'Access est toujours activée. Il existe un compte "Administrateur" possédant tous les droits et n'ayant pas de mot de passe. Pour activer la sécurité sur Access on procède en général de la manière suivante :

- On donne un mot de passe à l'utilisateur "Administrateur".
- On crée un nouveau compte d'administrateur, avec un autre nom de préférence ("Admin" par exemple) auquel on donne tous les droits et propriétés
- On supprime le compte "Administrateur" du groupe "Administrateurs". Il n'est en effet pas possible de supprimer les Groupes et Utilisateurs par défaut.

La dernière étape n'est pas indispensable, par contre la première l'est, car sans cela, n'importe qui ayant Access sur son poste pourra ouvrir votre base de données.

## **Utilisateurs et groupes**

C'est une notion classique de la sécurité des SGBD. Un groupe représente l'ensemble des utilisateurs possédant les mêmes droits. Un utilisateur représente en générale une personne.

Chaque utilisateur appartient au minimum au groupe des utilisateurs, mais il peut appartenir à plusieurs groupes.

Un utilisateur hérite toujours des droits du(des) groupe(s) dont il dépend, mais peut avoir des droits modifiés par rapport à celui-ci. Attention, en cas de contradiction entre les droits du groupe et ceux de l'utilisateur, c'est toujours la condition la **moins** restrictive qui l'emporte.

Pour simplifier la gestion des droits, on définit habituellement les groupes puis les utilisateurs.

Lorsqu'un groupe est propriétaire, chaque utilisateur du groupe est également propriétaire.

L'héritage des droits d'un groupe est paramétrable.

## **Héritage des objets**

Les droits d'un groupe / utilisateur se définissent normalement pour chaque objet.

Pour une base de données contenant de multiples objets, vous voyez immédiatement le code que cela peut engendrer. Il est possible de propager les droits d'un objet à tous les objets que celui-ci contient. Une solution consiste donc à donner des restrictions standards à l'objet base de données puis à adapter au cas par cas sur les objets. Attention, cette méthode n'est pas réversible, elle n'est donc pas sans risque. Cet héritage n'est jamais transverse, mais il est possible de définir des droits pour une catégorie générique d'objet.

## Objet Group

En soit un objet Group n'est pas très complexe. Il ne possède que la collection Users qui représente les utilisateurs membres du groupe, une propriété **Name** qui est son nom et deux méthodes. C'est celles-ci que nous allons voir en détails.

### SetPermissions

Attribue les permissions sur un objet pour un groupe ou un utilisateur. De la forme  
`Group.SetPermissions Name, ObjectType, Action, Rights [, Inherit] [, ObjectTypeId]`



Il s'agit là d'une méthode. Contrairement à la valorisation des propriétés ADOX qui se fait **avant** l'ajout de l'objet à la collection, on applique les méthodes à l'objet **seulement après** son ajout à la collection.

Passons en revue ces paramètres, mais notez bien qu'il s'appliquent aussi bien au groupe qu'à l'utilisateur. Les différences seront vues dans l'étude de l'objet User.

#### ObjectType

Le paramètre ObjectType peut prendre une des valeurs suivantes :

Constante	Valeur	Description
adPermObjProviderSpecific	-1	Défini un objet spécifique du fournisseur.. Une erreur se produira si le paramètre ObjectType n'est pas fourni
adPermObjTable	1	Objet Table.
adPermObjColumn	2	objet Column
adPermObjDatabase	3	L'objet est la base de données
adPermObjProcedure	4	Objet Procedure.
AdPermObjView	5	Objet View

On retrouve la tous les objets du modèle plus l'objet Database. La valeur adPermObjProviderSpecific permet d'utiliser des objets spécifiques au fournisseur, par exemple pour Access les états ou les formulaires.

#### Name

Donne le nom de l'objet cible, si vous ne précisez pas ce nom vous devez passer NULL mais vous aurez une erreur si vous omettez le paramètre. Dans ce cas, les droits seront attribués à tous les **nouveaux** objets de même type contenu dans le catalogue (catégorie générique).

Pour l'objet database, vous devez donner une chaîne vide ("" ) au paramètre Name. Voir dans l'exemple plus loin.

#### Action

Ce paramètre est combiné avec le paramètre Rights. Il peut prendre les valeurs suivantes :

Constante	Valeur	Description
adAccessGrant	1	Le groupe aura au moins les autorisations demandées.
adAccessSet	2	Le groupe aura exactement les autorisations demandées
adAccessDeny	3	Le groupe n'aura pas les autorisations demandées.
adAccessRevoke	4	Tous les droits d'accès explicites que possèdent le groupe seront annulés.

Détaillons ces valeurs car elles dépendent fortement de la stratégie choisie ainsi que de la gestion de l'existant.

Lorsque l'on considère un utilisateur, on entend par droits (autorisations) explicites, les droits qui appartiennent à celui-ci et par droits implicites les droits de son groupe. Pour un groupe, tous les droits sont explicites.

adAccessRevoke retire donc tous les droits du groupe, il n'est pas utile de renseigner le paramètre Rights (évidemment). Le fait de retirer les droits d'un groupe ne retire pas les droits d'un utilisateur appartenant au groupe lorsqu'ils sont différents de ceux du groupe.

adAccessDeny enlève tous les droits définis dans le paramètre Rights.

adAccessSet est la valeur que l'on utilise en général lors de la création du fichier de sécurité. Les droits définis dans le paramètre Rights seront exactement les droits donnés, ceux existant précédemment seront remplacés ou modifiés.

AdAccessGrant est la valeur pour la modification de droits. Au lieu de redéfinir toute la liste des droits on précise le droit que l'on souhaite modifier, les autres droits ne seront pas redéfinis.

### Rights

Valeur de type Long représentant un masque binaire des droits pouvant être une ou plusieurs des valeurs suivantes. Certaines de ces valeurs sont exclusives (adRightFull par exemple) :

Constante	Valeur	Description
adRightCreate	16384 (&H4000)	Le groupe à l'autorisation de créer des objets de ce type
adRightDelete	65536 (&H10000)	Le groupe a l'autorisation de supprimer des données de l'objet
adRightDrop	256 (&H100)	Le groupe a l'autorisation de supprimer l'objet
adRightExclusive	512 (&H200)	Le groupe a l'autorisation d'accéder de façon exclusive à l'objet
adRightExecute	536870912 (&H20000000)	Le groupe a l'autorisation d'exécuter l'objet.(macro Access par exemple)
adRightFull	268435456 (&H10000000)	Le groupe a toutes les autorisations concernant l'objet
adRightInsert	32768 (&H8000)	Le groupe a l'autorisation d'insérer des données dans l'objet.
adRightMaximumAllowed	33554432 (&H2000000)	Le groupe a le maximum d'autorisations spécifiques autorisées par le fournisseur.
adRightNone	0	Le groupe n'a aucune autorisations concernant l'objet.
adRightRead	-2147483648 (&H80000000)	Le groupe a l'autorisation de lire l'objet
adRightReadDesign	1024 (&H400)	Le groupe a l'autorisation de lire la définition de l'objet
adRightReadPermissions	131072 (&H20000)	Le groupe a l'autorisation de lire les autorisations de l'objet
adRightReference	8192 (&H2000)	Le groupe a l'autorisation de référencer l'objet
adRightUpdate	1073741824 (&H40000000)	Le groupe a l'autorisation de modifier des données dans l'objet.
adRightWithGrant	4096 (&H1000)	Le groupe a l'autorisation d'accorder des autorisations concernant l'objet
adRightWriteDesign	2048 (&H800)	Le groupe a l'autorisation de modifier la structure de l'objet
adRightWriteOwner	524288 (&H80000)	Le groupe a l'autorisation de modifier le propriétaire de l'objet
adRightWritePermissions	262144 (&H40000)	Le groupe a l'autorisation de modifier des autorisations spécifiques de l'objet.

Comme il s'agit de masque binaire, on compose les droits avec l'opérateur "OR".

Une erreur fréquente consiste à considérer l'attribution des droits par programmation comme fonctionnant à l'identique de l'assistant sécurité. Celui-ci crée implicitement des droits lorsque l'on attribue certains droits à l'utilisateur (par exemple l'autorisation "modifier les données" donne automatiquement l'autorisation "lire les données"). Il n'en est pas de même par programmation. Vous devez définir de façon cohérente et explicite les droits sous peine de bloquer vos utilisateurs.

Une autre erreur consiste à utiliser un droit qui n'existe pas pour l'objet considéré. Ainsi j'ai déjà vu donner le droit adRightExecute sur des procédures, sûrement parce que l'on dit exécuter une requête, alors que c'est le droit adRightRead qu'il faut utiliser.

### Inherit

Paramètre optionnel qui détermine comment les objets contenus dans l'objet cible hériteront des autorisations. Les valeurs peuvent être :

Constante	Valeur	Description
adInheritNone	0	Valeur par défaut. Pas d'héritage.
adInheritObjects	1	Les objets non-conteneurs héritent des autorisations.
adInheritContainers	2	Les objets conteneurs héritent des autorisations.
adInheritBoth	3	Tous les objets héritent des autorisations.
adInheritNoPropagate	4	Les objets n'ayant pas déjà hérités d'autorisations hériteront.

L'héritage a parfois des résultats surprenants avec ADOX, je vous conseille donc de ne pas en abuser. Toutefois on peut toujours déterminer une configuration type par groupe et la donner à la base de données avec un héritage, afin que les objets contenus héritent des autorisations. Je ne vous conseille pas d'utiliser cette technique.

### ObjectTypeId

Ce paramètre est obligatoire avec le type adPermObjProviderSpecific. Cela permet d'accéder à certains objets contenus dans le SGBD. Il faut passer comme valeur le GUID de l'objet. Par exemple pour Access les GUIDs suivants permettent de définir des autorisations pour les formulaires, états et macro

Object	GUID
Form	{c49c842e-9dcb-11d1-9f0a-00c04fc2c2e0}
Report	{c49c8430-9dcb-11d1-9f0a-00c04fc2c2e0}
Macro	{c49c842f-9dcb-11d1-9f0a-00c04fc2c2e0}

### GetPermissions

Permet de lire les autorisations d'accès d'un objet. Ce n'est pas toujours très simple à interpréter par programmation, aussi je vous conseille plutôt de supprimer puis de recréer les autorisations.

De la forme

*ReturnValue* = *Group*.GetPermissions(*Name*, *ObjectType* [, *ObjectTypeId*])

Où *ReturnValue* est une valeur de type Long représentant le masque binaire. Pour tester si une autorisations existe dans le masque utiliser l'opérateur AND

La fonction suivante permet d'afficher les droits dans la fenêtre d'exécution.

```
Private Sub EcrisDroit(Droits As Long)

    If Droits And adRightCreate Then Debug.Print "Création"
    If Droits And adRightDelete Then Debug.Print "Effacement"
    If Droits And adRightDrop Then Debug.Print "Suppression"
    If Droits And adRightExclusive Then Debug.Print "Exclusif"
    If Droits And adRightExecute Then Debug.Print "Exécution"
    If Droits And adRightInsert Then Debug.Print "Insertion"
    If Droits And adRightRead Then Debug.Print "Lecture D"
    If Droits And adRightReadDesign Then Debug.Print "Lecture S"
    If Droits And adRightReadPermissions Then Debug.Print "LirePermission"
    If Droits And adRightReference Then Debug.Print "Reference"
    If Droits And adRightUpdate Then Debug.Print "Modification"
    If Droits And adRightWithGrant Then Debug.Print "ModifPermission"
    If Droits And adRightWriteDesign Then Debug.Print "Modif Structure"
    If Droits And adRightWriteOwner Then Debug.Print "Modif Prop"

End Sub
```

Avec un appel de procédure comme :

```
temp = MonGrou.GetPermissions("tblAnomalie", adPermObjTable)
Call EcrisDroit(temp)
```

## Objet User

Cet objet ressemble fortement à l'objet Group dans sa programmation. Il possède une méthode de plus.

### ChangePassword

Permet de modifier le mot de passe d'un utilisateur.

*User.ChangePassword OldPassword, NewPassword*

Là pas besoin d'explication si ce n'est qu'un des paramètres peut être la chaîne vide ""

### GetPermissions & SetPermissions

S'utilise comme pour l'objet Group.

Comme je l'ai déjà dit un utilisateur membre d'un groupe peut avoir des autorisations différentes de celles du groupe. Pour des raisons de maintenance, cette situation est toutefois à proscrire. N'oubliez pas que sur un mélange de propriétés utilisateur / groupe c'est toujours l'autorisation la moins restrictive qui l'emporte.

### Properties

Les objets User et Group ont une collection properties.

## Exemple

Dans l'exemple suivant nous allons sécuriser une base de données, créer deux groupes et deux utilisateurs.

En phase de création de base, commencez toujours par gérer la sécurité comme vous allez le voir c'est beaucoup plus simple.

```
Dim Catalogue As ADOX.Catalog, MonGrou As ADOX.Group, strconn As String

Set Catalogue = New ADOX.Catalog
strconn = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" &
"D:\User\jmarc\tutorial\ADOX\Test\NouvBase1.mdb" & ";Jet OLEDB:System
database = D:\User\jmarc\tutorial\ADOX\system.mdw;User Id=Admin; Password="
Catalogue.Create strconn
Catalogue.Users("Admin").ChangePassword "", "password"
Catalogue.Groups.Append "Utilisateurs"
Set MonGrou = Catalogue.Groups("Utilisateurs")
With MonGrou
    .SetPermissions "", adPermObjDatabase, adAccessSet, adRightRead,
adInheritNone
    .SetPermissions Null, adPermObjTable, adAccessSet, adRightInsert Or
adRightRead Or adRightUpdate, adInheritNone
    .SetPermissions Null, adPermObjProcedure, adAccessSet, adRightRead
Or adRightReadDesign, adInheritNone
    .SetPermissions Null, adPermObjView, adAccessSet, adRightRead Or
adRightReadDesign, adInheritNone
End With
Catalogue.Groups.Append "Consultants"
Set MonGrou = Catalogue.Groups("Consultants")
With MonGrou
    .SetPermissions "", adPermObjDatabase, adAccessSet, adRightNone,
adInheritNone
    .SetPermissions Null, adPermObjTable, adAccessSet, adRightNone,
adInheritNone
    .SetPermissions Null, adPermObjProcedure, adAccessSet, adRightNone,
adInheritNone
    .SetPermissions Null, adPermObjView, adAccessSet, adRightRead,
adInheritNone
End With
Catalogue.Users.Append "Emile", ""
Catalogue.Users("Emile").ChangePassword "", "xtf22re"
Catalogue.Groups("Utilisateurs").Users.Append "Emile"
```

```

With Catalogue.Users("Emile")
    .SetPermissions "", adPermObjDatabase, adAccessSet, adRightRead,
adInheritNone
    .SetPermissions Null, adPermObjTable, adAccessSet, adRightInsert Or
adRightRead Or adRightUpdate, adInheritNone
    .SetPermissions Null, adPermObjProcedure, adAccessSet, adRightRead
Or adRightReadDesign, adInheritNone
    .SetPermissions Null, adPermObjView, adAccessSet, adRightRead Or
adRightReadDesign, adInheritNone
End With
Catalogue.Users.Append "Leon", ""
Catalogue.Users("Leon").ChangePassword "", "pved58t"
Catalogue.Groups("Consultants").Users.Append "Leon"
With Catalogue.Users("Leon")
    .SetPermissions "", adPermObjDatabase, adAccessSet, adRightNone,
adInheritNone
    .SetPermissions Null, adPermObjTable, adAccessSet, adRightNone,
adInheritNone
    .SetPermissions Null, adPermObjProcedure, adAccessSet, adRightNone,
adInheritNone
    .SetPermissions Null, adPermObjView, adAccessSet, adRightRead,
adInheritNone
End With

```

Comme nous le voyons, la première étape après la création est de changer le mot de passe de l'administrateur afin d'activer la sécurité.

Je crée ensuite deux groupes dont je définis les droits puis deux utilisateurs (1 par groupe). Notez que je redéfinis les mêmes droits puisque ADO avec Access ne génère pas l'héritage implicite des droits du groupe.

Dans ce cas, je n'ai encore créé aucun objet dans ma table. Ceci me permet d'utiliser des catégories génériques pour l'attribution des droits (utilisation de Null pour le paramètre Name).

Vous remarquerez enfin que j'utilise SetPermissions après l'ajout de l'objet à sa collection

## ***Techniques de sécurisation***

Il est bien évident que gérer la sécurité par le code n'est pas une sinécure, aussi on essaye toujours de passer par les assistants ad hoc des SGBD plutôt que de se lancer dans un code aventureux.

Dans le cas d'Access, il est beaucoup plus parlant de gérer les sécurités depuis Access. Si toutefois vous devez utiliser le code, vous gagnerez en efficacité et en simplicité à suivre le modèle DAO pour gérer la sécurité, du moins tant que Microsoft n'aura pas sorti une version ADO faisant cela efficacement.

Nous allons partir du principe dans cette partie que l'emploi du code avec ADO est indispensable.

Grosso modo il n'existe que deux cas, soit vous allez modifier des groupes, des utilisateurs ou des autorisations existantes soit il vous faut tout créer.

## **Modification**

Si la sécurité est déjà bien gérée, il ne s'agit que d'ajouter des utilisateurs et de bien gérer les droits d'accès ou de propriété. La technique consiste toujours à créer un utilisateur dans un groupe existant, afin de ne pas avoir à gérer tous ses droits objet par objet. Lors de la modification d'autorisations, il est souvent plus simple de détruire les anciennes, puis de ré attribuer les autorisations. Cherchez toujours à donner la propriété des nouveaux objets à l'administrateur ou alors créez un utilisateur fantôme, appelés par exemple "Code", qui possède les autorisations nécessaires à tous vos accès par le code.

Si la sécurité est mal gérée, détruisez tout et repartez du début.

Le code suivant vous donne un exemple de la ré attribution de la propriété des tables.

```
Private Sub Command9_Click()  
  
Dim cnn1 As ADODB.Connection  
Dim Catalogue As ADOX.Catalog, MaTable As ADOX.Table  
  
Set Catalogue = New ADOX.Catalog  
Set cnn1 = New ADODB.Connection  
With cnn1  
    .Provider = "Microsoft.Jet.OLEDB.4.0;"  
    .ConnectionTimeout = 30  
    .CursorLocation = adUseClient  
    .IsolationLevel = adXactChaos  
    .Mode = adModeShareExclusive  
    .Properties("Jet OLEDB:System database") =  
"D:\User\jmarc\tutorial\ADOX\system.mdw"  
    .Open "Data Source=D:\User\jmarc\tutorial\ADOX\Test\NouvBase1.mdb  
;User Id=Admin; Password=password"  
End With  
Catalogue.ActiveConnection = cnn1  
For Each MaTable In Catalogue.Tables  
    If StrComp(Catalogue.GetObjectOwner(MaTable.Name, adPermObjTable),  
"admin", vbTextCompare) <> 0 And  
StrComp(Catalogue.GetObjectOwner(MaTable.Name, adPermObjTable), "engine",  
vbTextCompare) <> 0 Then  
        Catalogue.SetObjectOwner MaTable.Name, adPermObjTable, "Admin"  
    End If  
Next  
  
End Sub
```

Comme vous le voyez dans le test, n'essayez pas de prendre la propriété des tables dont le propriétaire est "engine" (information de schéma du moteur Jet) vous déclencheriez une erreur.

## **Création**

Lors de la création de la base, je vous conseille vivement de créer la sécurité en premier. Vous pouvez dès lors utiliser des objets génériques et définir vos droits assez rapidement. Il vaut toujours mieux avoir un groupe de trop que l'on supprime à la fin, plutôt que de se rendre compte qu'il en manque un.

Définissez votre système de sécurité avant de coder. Par le code il vaut mieux multiplier les groupes que les utilisateurs.

## **Une autre solution : le DDL**

A partir d'Access 2000, vous pouvez contourner les faiblesses de ADO pour la création des utilisateurs et des groupes en utilisant le DDL. Sachez que celui vous permet de gérer les PID.

Un objet Command peut être utilisé avec une requête de type :

```
CREATE USER Nom MotDePasse PID
```

Mais nous sortons là du cadre de cet article

## ***Conclusion sur la sécurité***

Comme nous l'avons vu, gérer la sécurité par le code n'est souvent pas un bon choix. Cela reste possible à faire dans certains cas particulier mais le code est relativement lourd et dur à corriger. Privilégiez toujours la solution des assistants tant que cela est possible.

Bien que je n'ai pas abordé cela, vous aurez une erreur si l'utilisateur qui se connecte n'a pas les droits d'écriture de structure et tente d'ouvrir le catalogue.

Ce catalogue peut être partiel en fonction des droits définis.

## **Conclusion**

Voilà nous avons parcouru le modèle objet ADOX dans son ensemble. Gardez bien en mémoire que celui-ci est fait pour intervenir sur la structure, pas seulement pour la lire, dans ce cas privilégiez OpenSchema.

Le modèle en lui-même est assez simple, mais pour pouvoir l'utiliser de façon approfondie vous devez étudier correctement votre fournisseur.

Bonne programmation.