Création d'un journal d'évènements sous Access 2010

Par Christophe WARIN

Date de publication : 7 septembre 2009

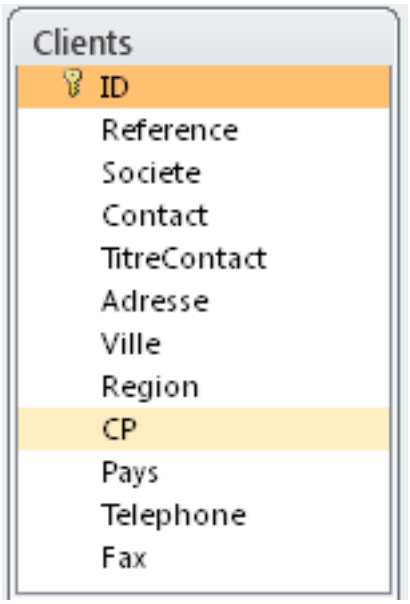
Cet article est une mise en pratique d'une nouvelle fonctionnalité d'Access 2010 : les évènements de tables.
Au sommaire : réalisation d'un journal d'évènements pour vos bases Access.

I - Introduction

La création d'un journal dans une base Access a toujours été une question récurrente sur les forums. Le premier obstacle rencontré jusqu'à Access 2010 était l'absence de trigger (déclencheur) sur les tables. Comment enregistrer le fait qu'une information ait été modifiée si aucun mécanisme ne signale ce changement ? Deuxième obstacle : la taille du journal. Avec une base de petite taille mais où les accès en écriture sont nombreux, le fichier journal atteint vite un volume monstrueux nécessitant une purge manuelle ou la mise en place d'un système de surveillance chargé de vider régulièrement les historiques obsolètes. Bien entendu, être obligé de coder un espion sur un autre est peu attractif. Heureusement, avec l'arrivée des évènements de tables que l'on peut aussi appeler déclencheurs ou trigger (à l'instar des SGBD plus robustes), Microsoft Access 2010 enfonce encore un peu plus le clou de l'application professionnelle.

Dans ce tutoriel, nous allons tenter de répondre au cahier des charges suivant :

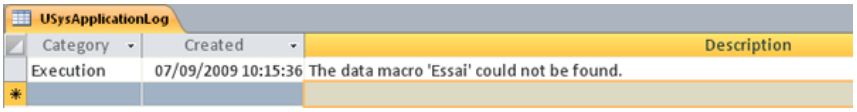
Enregistrer les modifications apportées à la table Clients ci-dessous :



Si vous n'avez jamais manipulé les macros de données (DataMacro), je vous invite à commencer par un autre article de découverte : ***Les nouveautés de Microsoft Access 2010 : Les triggers***

II - La table USysApplicationLog

Cette table est mentionnée dans quasiment tous les documents traitant des macros de données pour la simple et bonne raison que c'est ici que sont stockés les messages d'erreurs des évènements de tables ayant échoué.



Si vous ne voyez pas cette table, rendez-vous dans les options du panneau de navigation afin d'activer l'affichage des objets systèmes.

La méthode **LogEvent** des **DataMacros** offre au développeur la possibilité d'enrichir la table USysApplicationLog avec des évènements personnalisés. Par exemple, sur l'évènement **AfterUpdate** de la table **Clients** :

```
LogEvent
Description : Modification d'un enregistrement dans la table Clients.
```

L'attribut **Description** est de type texte. Pour y insérer une donnée provenant d'un champ de la table il est nécessaire de passer par une expression calculée :

```
LogEvent
Description : ="Modification d'un enregistrement dans la table Clients. IDClient=" & [ID]
```

Le sigle = placé en tête de l'attribut demande au moteur de base de données d'évaluer l'expression composée de deux variables : le texte et l'id du client.

Si cette solution a le mérite d'être très simple, elle reste cependant déconseillée puisque les évènements d'erreurs se retrouvent mélangés à un simple historique. D'un point de vue purement conceptuel il paraît anormal de stocker dans la même structure des données systèmes et des données métiers. Jamais il ne vous prendrait l'idée de stocker vos photos de vacances dans le répertoire System32 de Windows au beau milieu de centaines de dll.

USysApplicationLog

Category	Created	Description
User	07/09/2009 10:10:47	Modification d'un enregistrement dans la table Clients. IDClient=1
Execution	07/09/2009 10:15:36	The data macro 'Essai' could not be found.
User	07/09/2009 10:29:53	Modification d'un enregistrement dans la table Clients. IDClient=3
User	07/09/2009 10:29:57	Modification d'un enregistrement dans la table Clients. IDClient=16

III - Journalisation avancée

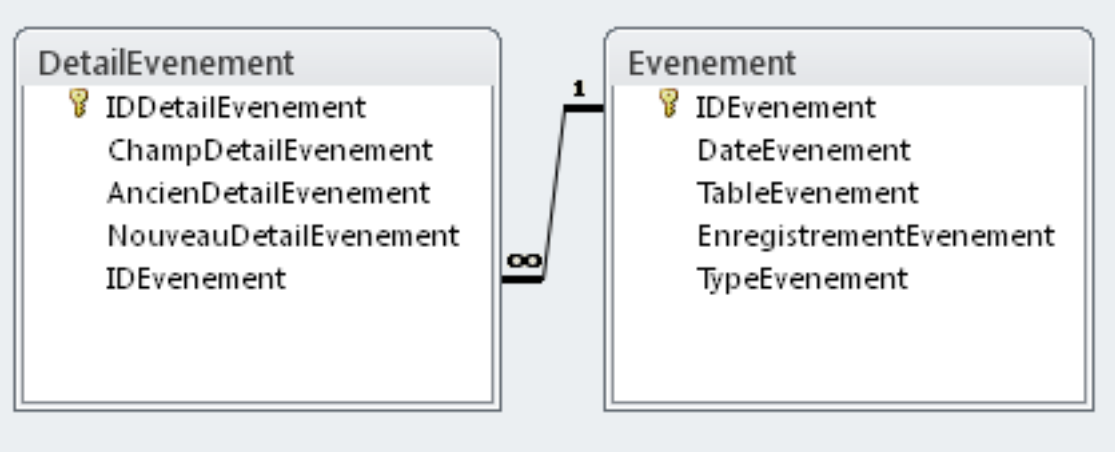
Comme la table **USysApplicationLog** semble insuffisante, nous allons créer notre propre structure de journalisation. Celle-ci est constituée de deux tables et permettra de lister les valeurs des champs avant et après mise à jour.

Table Evènement :

- IDEvenement : NuméroAuto - Clé primaire
  Identifiant de la table.
- DateEvenement : Date/Heure - Valeur par défaut : Now()
  Date où a eu lieu l'évènement
- TableEvenement : Texte
  Nom de la table où a eu lieu l'évènement
- EnregistrementEvenement : Numérique
  Identifiant de l'enregistrement qui a été modifié
- TypeEvenement : Texte
  Suppression, modification ou insertion

Table DetailEvenement :

- IDDetailEvenement : NuméroAuto - Clé primaire
  Identifiant de la table
- ChampDetailEvenement : Texte
  Nom du champ modifié
- AncienDetailEvenement : Texte
  Ancienne valeur
- NouveauDetailEvenement : Texte
  Nouvelle valeur
- IDEvenement : Numérique
  Clé étrangère correspondant à l'évènement dans la table Evenement



L'insertion d'un nouvel élément dans le journal se déroule en plusieurs étapes :

1 Création d'une nouvelle ligne dans la table **Evenement** dès qu'un client est mis à jour,
2 Création d'une nouvelle ligne dans la table **DetailEvenement** pour chaque champ concerné par cette modification.

III-A - Insertion dans la table Evenement

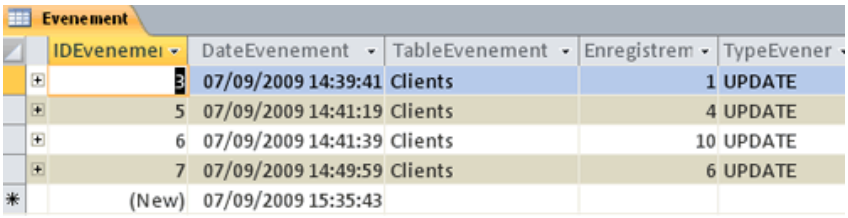
Pour que l'ajout d'une entrée dans le journal ait lieu à chaque modification d'un client, il faut se placer sur l'évènement **AfterUpdate** de la table **Clients**. On notera au passage que la valeur par défaut du champ nous soulage du traitement de la date.

```
Create a Record in Evenement
Alias RecordEvenement
  SetField
    Name : EnregistrementEvenement
    Value: [Clients].[ID]
  SetField
    Name : TableEvenement
    Value: "Clients"
  SetField
    Name : TypeEvenement
    Value: "UPDATE"
```

La syntaxe demande un peu de rigueur notamment en ce qui concerne la portée des membres. Dans le premier **SetField**, la valeur **[ID]** en remplacement de **[Clients].[ID]** aurait levé une erreur dans la table **USysApplicationLog** car le moteur s'attend à recevoir un champ de l'enregistrement **RecordEvenement** issu du bloc **CreateRecord**. Une autre solution serait de passer par une variable locale dans la racine de la **DataMacro** :

```
SetLocalVar
  Name : varIDClient
  Expr : [ID]
Create a Record in Evenement
Alias RecordEvenement
  SetField
    Name : EnregistrementEvenement
    Value: [varIDClient]
  SetField
    Name : TableEvenement
    Value: "Clients"
  SetField
    Name : TypeEvenement
    Value: "UPDATE"
```

Cette solution offre l'avantage d'afficher clairement en en-tête de macro les éléments qui seront réutilisés par la suite.



III-B - Insertion dans la table DetailEvenement

Pour chaque champ de la table **Clients**, la macro de données va devoir vérifier si celui-ci a été modifié ou non. Dans le cas où un changement a été repéré, une nouvelle ligne est ajoutée à la table **DetailEvenement** regroupant l'ancienne et la nouvelle valeur. Pour cela, deux fonctionnalités doivent être utilisées

- La fonction **Updated(nomduchamp)** qui permet de détecter si le champ passé en paramètre a été modifié.
- La syntaxe **Old.NomDuChamp** qui permet de récupérer l'ancienne valeur d'un champ modifié

Dans un premier temps, il est surtout question de la jointure entre les deux tables du journal. Il est nécessaire d'affecter l'identifiant de l'évènement en cours aux lignes qui vont être créées dans la table **DetailEvenement**. Le recours à une variable locale semble approprié à condition de placer l'appel de **SetLocalVar** dans le bloc **CreateRecord** sans quoi, et c'est encore un problème de portée, toute référence à l'enregistrement en cours de création sera perdue.

```
SetLocalVar
  Name : varIDClient
  Expr : [ID]
Create a Record in Evenement
Alias RecordEvenement
  SetField
    Name : EnregistrementEvenement
    Value: [varIDClient]
  SetField
    Name : TableEvenement
    Value: "Clients"
  SetField
    Name : TypeEvenement
    Value: "UPDATE"
  SetLocalVar
    Name : varIDEvenement
    Expr : [IDEvenement]
```

Cette formalité accomplie, il faut, pour chaque champ, répéter le même bloc d'instructions chargées d'alimenter la table **DetailEvenement** :

```
If Updated("reference") Then
  Create a Record In DetailEvenement
  Alias RecordDetail
    SetField
       Name : ChampDetailEvenement
       Value: "Reference"
    SetField
       Name : AncienDetailEvenement
       Value: [Old].[Reference]
    SetField
       Name : NoueavDetailEvenement
       Value: [Clients].[Reference]
    SetField
       Name : IdEvenement
       Value: [varIdEvenement]
End If
```

Bien entendu, il paraît fastidieux de devoir dupliquer ce bloc **If** pour l'ensemble des champs de la table **Clients**. Tout comme en VBA, il est possible de factoriser le code en isolant le bloc conditionnel dans une macro dédiée puis de faire appel à cette macro dans l'évènement **AfterUpdate** de la table.

Schématiquement, cela donne la macro **dm_detailEvenement** et les appels suivants :

```
dm_DetailEvenement(pNomChamp,pAncienneValeur,pNouvelleValeur,pID)
```

Les paramètres correspondent respectivement au nom du champ à tester, à son ancienne valeur, à sa nouvelle valeur et à l'identifiant de l'évènement courant.

Cette macro est ensuite appelée dans l'évènement de la table :

```
...
RunDataMacro  dm_DetailEvenement("Reference",[Old].[Reference],[Reference],[varIdEvenement])
RunDataMacro  dm_DetailEvenement("Societe",[Old].[Societe],[Societe],[varIdEvenement])
...
```

Soit en langage Macro :

dm_DetailEvenement :

```
Parameters
  pNomChamp
  pAncienneValeur
  pNouvelleValeur
  pID
If Updated(pNomChamp) Then
  Create a Record In DetailEvenement
  Alias RecordDetail
    SetField
       Name : ChampDetailEvenement
       Value: pNomChamp
    SetField
       Name : AncienDetailEvenement
       Value: pAncienneValeur
    SetField
       Name : NouveauDetailEvenement
       Value: pNouvelleValeur
    SetField
       Name : IdEvenement
       Value: pID
End If
```

***Attention** : la variable locale **varIdEvenement** du trigger **AfterUpdate** a une portée limitée à cet évènement et n'est pas disponible dans la nouvelle macro **dm_DetailEvenement**. Il est donc nécessaire de la passer en paramètre.*

*Pour créer une nouvelle macro de données, cliquez sur le bouton **Create Table Event** puis **Create Table Macro**.*

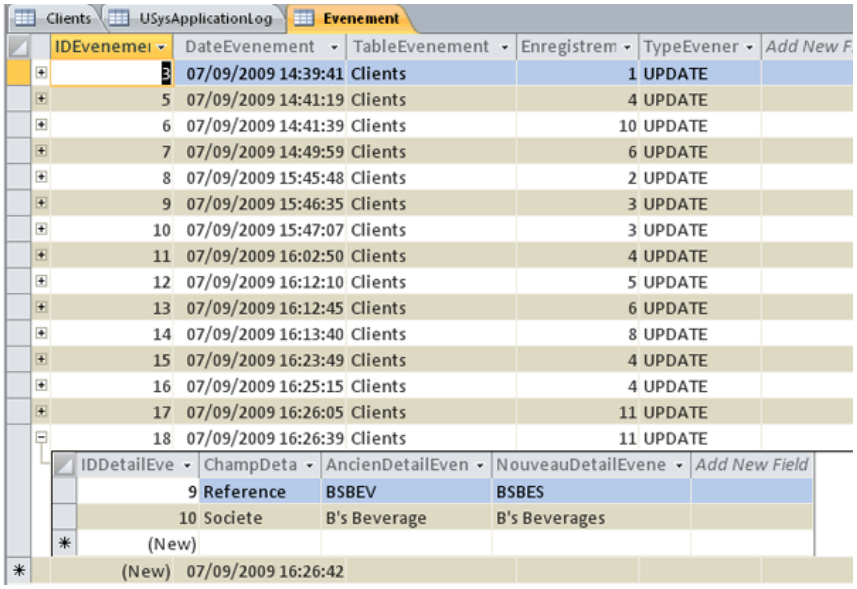
Clients.AfterUpdate :

```
SetLocalVar
  Name : varIDClient
  Expr : [ID]
Create a Record in Evenement
Alias RecordEvenement
  SetField
```

```
    Name : EnregistrementEvenement
    Value: [varIDClient]
  SetField
    Name : TableEvenement
    Value: "Clients"
  SetField
    Name : TypeEvenement
    Value: "UPDATE"
  SetLocalVar
    Name : varIDEvenement
    Expr : [IDEvenement]

RunDataMacro
  Macro Name : Clients.dm_DetailEvenement
  Where :
  Parameters
    pNomChamp : "Reference"
    pAncienneValeur : [Old].[Reference]
    pNouvelleValeur : [Reference]
    pID : varIdEvenement

RunDataMacro
  Macro Name : Clients.dm_DetailEvenement
  Where :
  Parameters
    pNomChamp : "Societe"
    pAncienneValeur : [Old].[Societe]
    pNouvelleValeur : [Societe]
    pID : varIdEvenement
...
```



IV - Limitation de la taille du journal

La journalisation est à présent opérationnelle et le journal va se remplir à vitesse grand V à tel point qu'il est primordial d'y appliquer un quota. Cet exemple se base sur une limitation du nombre d'évènements dans la table **Evenement**. L'idée est très simple, lorsque le journal est plein, on le vide par le bas à chaque insertion de sorte de garder un maximum de 10(,20, 100, …) évènements.

Pour cela, il est nécessaire de créer une DataMacro sur l'évènement **AfterInsert** de la table **Evenement** et d'activer conjointement la suppression en cascade sur la relation **Evenement-DetailEvenement** de telle sorte à propager la purge d'une table à l'autre.

La solution la plus facile serait de supprimer l'évènement le plus ancien à chaque nouvelle insertion dès que le quota est atteint ; il s'agit d'une pile. Seul bémol, en cas de redimensionnement du quota, le mécanisme est complètement déstabilisé. Le choix le plus souple consiste à lister dans une requête les **IDEvenement** en ordre décroissant et à supprimer les superflus à l'aide d'un compteur (**varN**). Dans le cas d'un journal à fort volume, le système de pile reste bien entendu préférable car bien moins gourmand en ressources.

```
SetLocalVar
  Name : varN
  Expr : 10

For Each Record In qry_Evenement_DESC;
  Where :
  Alias : RecordEvenement
  SetLocalVar
    Name : varN
    Expr : [varN]-1
  If [varN]<0 Then
    DeleteRecord
      RecordAlias : RecordEvenement
  End If
```

Code de la requête qry_Evenement_DESC :

```
SELECT IdEvenement
FROM Evenement
ORDER BY IDEvenement DESC;
```

En créant une table **Parametre(NomParametre,ValeurParametre)** contenant l'enregistrement **(QuotaJournal,10)** il est tout à fait possible de rendre le journal évolutif : l'utilisateur n'aura qu'à modifier le contenu de la table pour mettre à jour son quota :

```
Look Up A Record In Parametre
  Where : "NomParametre='QuotaJournal'"
  Alias : RecordParametre

  SetLocalVar
    Name : varN
    Expr : [ValeurParametre]

For Each Record In SELECT IdEvenement FROM Evenement ORDER BY IDEvenement DESC;
  Where :
  Alias : RecordEvenement
  SetLocalVar
    Name : varN
    Expr : [varN]-1
  If [varN]<0 Then
    DeleteRecord
      RecordAlias : RecordEvenement
  End If
```

V - Conclusion

Je dois l'avouer, j'étais un peu réticent aux évènements de table lorsque je les ai découverts pour la première fois. Le système de macro me semblait particulièrement lourd et peu attractif. Toujours est-il que l'on y prend goût, le copier-coller de bloc est possible, la factorisation en " sous-macro " aussi. Bref : une nouveauté à utiliser absolument !