Créer une application de grille sans Excel en C#

Ce mois-ci, la rubrique « Programmation Windows » ne se code pas en C++ mais en C#. Je vous promets du Level 300 car il y a des threads, de l'accès aux données SQL Server, des appels Web Services WCF et un Timer d'affichage pour le thread principale de l'application.

Christophe PICHAUD | .NET Rangers by Sogeti
Consultant sur les technologies Microsoft
christophepichaud@hotmail.com | www.windowscpp.net

Le sujet de cet article est de créer une application qui représente une feuille Excel avec des données issues de source hétérogènes, des formules calculées mais sans Excel ! Oui, on va tout coder à la main.

Choisir sa grille

J'entends déjà au loin les sirènes des acheteurs de grilles qui vont s'engouffrer pour me dire d'aller acheter la grille de Telerik, SyncFusion ou Component One... Désolé, le Framework .NET est suffisamment riche pour nous offrir des grilles de très grande qualité et entièrement customisables. Je veux pouvoir choisir la dimension d'une cellule, choisir une police de caractère, choisir une couleur d'arrière-plan ou principale. Et donc en choisissant de rester avec la grille standard du Framework .NET, je suis certain de trouver toute la documentation nécessaire à la construction de mon application. Je n'aurai pas besoin de me balader dans 250 samples de Grilles en cherchant comment faire telle ou telle fonctionnalités... Comme dirait Bill, « Simple is beautiful ». Je veux une rapidité totale d'exécution et en aucun cas un mécanisme de binding évolué. Je peux pouvoir décrire mes colonnes et coder le remplissage des cellules à la main. Et pour faire cela, le contrôle DataGridView fera l'affaire.

Architecture WinForms ou WPF ?

Sur codeplex.com, il existe une DataGrid dans le projet WPFToolkit mais c'est pour WPF. Je n'ai pas besoin de WPF et de ses mécanismes de binding one-way two-way ou both, je veux du WinForms car c'est performant et qu'il n'y a jamais de trucs tordu. Tout ce qui doit arriver arrive en WinForms ; ce n'est pas le cas du XAML de WPF. Et puis Microsoft ne fabrique pas ses applications en WPF et nous savons pourquoi... Donc voilà, nous allons partir sur une application WinForms et la grille DataGridView du .NET Framework.

L'application à réaliser

Voici à quoi ressemble ce que nous allons coder : Fig.1.
Je vais faire un zoom sur cette grille qui est moyennement compliquée :

- Il y a des accès aux données SQL Server ;
- Il y a des appels à un Web service Reuters ;
- Il y a des formules calculées ;
- Il y a un thème graphique.

Par où commence-t-on ?

Tout d'abord, il faut déposer un composant DataGridView sur la Form. Ensuite, on clique sur la petite icône de la grille qui permet de gérer les colonnes. On se retrouve dans l'éditeur de colonnes. Fig.2.
Je positionne 4 choses pour chaque colonne :

- Le header text : le contenu de la cellule ;
- Le name : le nom de la cellule d'entête -> attention, cela sera déclaré dans le fichier de code du designer ;
- Le sort mode : je positionne à NotSortable ;
- La Width : je positionne à 50 ou 100, pas plus.

Une fois que la grille possède une définition de ses colonnes, je vais passer dans le code pour déclarer ses colonnes avec une énumération. Eh oui, dans le code, on va dire je veux la colonne « Devise » et non la colonne 8... On fait les choses bien ; même si on est en C#.

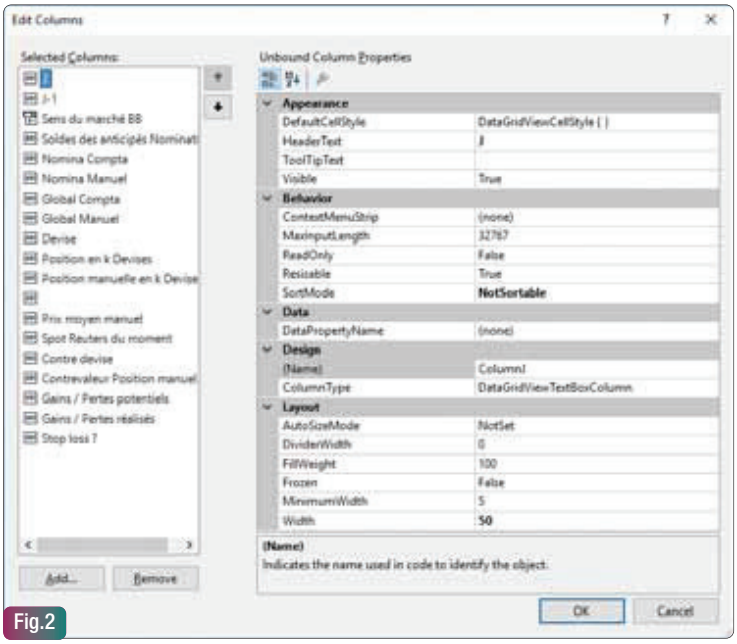
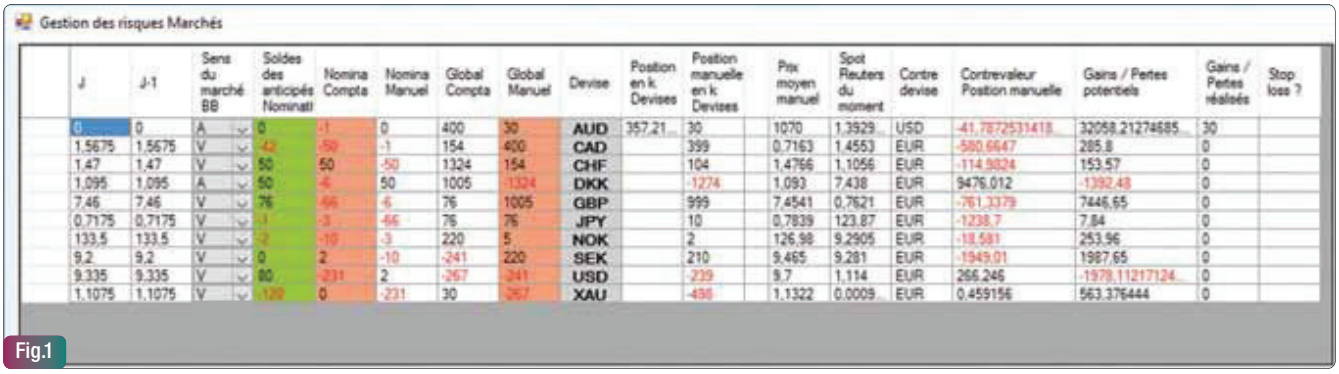


Fig.2



	J	J-1	Sens du marché BB	Soldes des anticipés Nominati	Nomina Compta	Nomina Manuel	Global Compta	Global Manuel	Devise	Position en k Devises	Position manuelle en k Devises	Prix moyen manuel	Spot Reuters du moment	Contre devise	Contrevaleur Position manuelle	Gains / Pertes potentiels	Gains / Pertes réalisés	Stop loss ?
	0	0	A	0	-1	0	400	30	AUD	357,21	30	1070	1,3929	USD	-41,7872531418	32058,21274685	30	
	1,5675	1,5675	V	-42	-50	-1	154	400	CAD		399	0,7163	1,4553	EUR	-580,6647	285,8	0	
	1,47	1,47	V	50	50	-50	1324	154	CHF		104	1,4766	1,1056	EUR	-114,9824	153,57	0	
	1,095	1,095	A	50	-6	50	1005	-1324	DKK		-1274	1,093	7,438	EUR	9476,012	-1392,43	0	
	7,46	7,46	V	76	-66	-6	76	1005	GBP		999	7,4541	0,7621	EUR	-761,3379	7446,65	0	
	0,7175	0,7175	V	-1	-3	-66	76	76	JPY		10	0,7839	123,87	EUR	-1238,7	7,84	0	
	133,5	133,5	V	-2	-10	-3	220	5	NOK		2	126,98	9,2905	EUR	-18,581	253,96	0	
	9,2	9,2	V	0	2	-10	-241	220	SEK		210	9,465	9,281	EUR	-1949,01	1987,65	0	
	9,335	9,335	V	80	-231	2	-267	-241	USD		-239	9,7	1,114	EUR	266,246	-1978,11217124	0	
	1,1075	1,1075	V	-120	0	-231	30	-267	XAU		-496	1,1322	0,0009	EUR	0,459156	563,376444	0	

Fig.1

```
public enum RowEnum : int
{
  J = 0,
  JMoins1 = 1,
  SensDuMarchéBB = 2,
...
  GainsPertesRealises = 17,
  StopLoss = 18
}
```

Maintenant que la grille est définie ainsi que la position des colonnes, il ne reste plus qu'à définir la classe qui va représenter une ligne de la grille. C'est le même code que l'énumération mais dans une classe :

```
public class DataRowNew
{
  public double J;
  public double JMoins1;
  public string SensDuMarchéBB;
...
  public double GainsPertesRealises;
  public string StopLoss;
}
```

Voilà tout ce dont j'ai besoin pour commencer à alimenter la grille... Maintenant je vous dévoile l'astuce pour que la grille prenne forme. Nous allons recopier l'ensemble des données de la feuille Excel que nous souhaitons remplacer dans la grille. Ensuite, nous allons appuyer sur un bouton qui sérialise la grille sous forme de liste d'éléments de ligne. Nous allons avoir une liste de DataRowNew dans un fichier XML. Nous allons aussi coder la méthode de désérialisation pour charger le fichier XML au démarrage de l'application.

Load et Save

Le code n'est pas très compliqué ; pour Load, on dépile la liste de DataRowNew du XML et on remplit chaque cellule une par une...

```
public void Load(string fileName, DataGridView dataGridView)
{
  List<DataRowNew> dataRows = new List<DataRowNew>();
  XmlSerializer xmls = new XmlSerializer(typeof(List<DataRowNew>));
  StreamReader sr = new StreamReader(fileName);
  dataRows = (List<DataRowNew>)xmls.Deserialize(sr);
  sr.Close();

  int rows = dataRows.Count-1;
  dataGridView.Rows.Clear();
  dataGridView.Rows.Add(rows);

  int count = 0;
  foreach (DataRowNew row in dataRows)
  {
    DataGridViewRow r = dataGridView.Rows[count];
    r.Height = 15;

    dataGridView.Rows[count].Cells[0].Value = row.J;
    dataGridView.Rows[count].Cells[1].Value = row.JMoins1;
    dataGridView.Rows[count].Cells[2].Value = row.SensDuMarchéBB;
...
    dataGridView.Rows[count].Cells[17].Value = row.GainsPertesRealises;
    dataGridView.Rows[count].Cells[18].Value = row.StopLoss;

    count++;
  }
}
```

La méthode Save fait le contraire :

```
public void Save(string fileName, DataGridView dataGridView)
{
  List<DataRowNew> dataRows = new List<DataRowNew>();

  foreach (DataGridViewRow row in dataGridView.Rows)
  {
    int count = 0;
    DataRowNew dataRow = new DataRowNew();
    foreach (DataGridViewCell cell in row.Cells)
    {
      switch (count)
      {
        case 0:
          dataRow.J = GridHelper.GetDoubleValue(row, RowEnum.J);
          break;
...
        case 18:
          dataRow.StopLoss = GridHelper
.GetStringValue(row, RowEnum.StopLoss);
          break;
      }

      count++;
    }
    dataRows.Add(dataRow);
  }

  XmlSerializer xmls = new XmlSerializer(typeof(List<DataRowNew>));
  StreamWriter sw = new StreamWriter(fileName);
  xmls.Serialize(sw, dataRows);
  sw.Close();
}
```

Maintenant, on met deux boutons sur la Form pour faire les fonctionnalités de Load & Save. Ainsi, on lance l'application et la grille est vierge. On prend son temps et on recopie toutes les données de la feuille Excel dans la DataGridView .NET. Une fois que c'est terminé, on clique sur le bouton Save et on obtient un fichier XML :

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfDataRowNew xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:
xsd="http://www.w3.org/2001/XMLSchema">
 <DataRowNew>
  <J>0</J>
  <JMoins1>0</JMoins1>
  <SensDuMarchéBB>A</SensDuMarchéBB>
  <SoldeDesAnticipésNominatif>0</SoldeDesAnticipésNominatif>
...
  <GainsPertesPotentiels13>32059.154472969421</GainsPertesPotentiels13>
  <GainsPertesRealises>30</GainsPertesRealises>
  <StopLoss />
```

```
</DataRowNew>
<DataRowNew>
...
```

Au chargement de l'application, on appelle la méthode Load() et la grille est automatiquement remplie... Magic. Ok, la grille est moche. C'est blanc et noir... On va pouvoir attaquer les choses sérieuses...

L'accès aux données SQL Server

Certaines colonnes vont être remplies par des appels SQL vers SQL Server toutes les 3 secondes. Donc on fait une couche DAL standard. On va faire des requêtes SQL et on va récupérer les données dans des classes C#. Oui Monsieur, à l'ancienne. Pour faciliter l'ouverture des connexions, la fermeture, les accès aux données en général, je vais utiliser le DAAB V2. C'est le Data Access Application Block 2.0 ; c'est en téléchargement sur le Microsoft Download Center. C'est une couche simple qui permet de faire rapidement des appels à SQL Server. C'est la version qui est sortie juste avant le Data Application Block de Enterprise Library... C'est du sans surprise. Allez-y ! On passe la connexion string et c'est tout. On s'applique à fermer le data reader et le tour est joué.

```
public static void LoadNominaCompta()
{
  DB.NominaComptaCollection.Clear();

  string cnx = Settings.GetConnectionString();
  SqlDataReader dr = SqlHelper.ExecuteReader(cnx, CommandType.Text,
"SELECT CODE_DEVISE, POSITION FROM NOMINA_COMPTA");
  while (dr.Read())
  {
    string devise = Convert.ToString(dr["CODE_DEVISE"]);
    double position = Convert.ToDouble(dr["POSITION"]);

    NominaCompta nc = new NominaCompta();
    nc.Devise = devise;
    nc.Position = position;
    DB.NominaComptaCollection.Add(nc);
  }
  dr.Close();
}
```

On remplit une collection en mémoire et il va falloir la charger dans la grille. Continuons de récupérer les données de sources hétérogènes...

L'accès aux données Web Services Reuters

Pour aller chercher le cours Reuters, j'ai besoin d'un client proxy vers le fichier SVC. Bouton droit « Add Service Reference », je rentre l'URL du service et Visual Studio me génère la référence de service. La méthode CallReutersWS itère dans un dictionnaire de <devise, reutersData>. En effet, avant d'appeler le WS, je dois savoir sur quelles devises je fais une interrogation. Donc, j'ai une routine qui récupère cette information dans la grille.

```
public static void CallReutersWS()
{
  lock (Svc.LockReuters)
  {
    Svc.ReutersDataCollection.Clear();

    foreach (KeyValuePair<string, ReutersData> kvp
in ReutersDataCollectionPrepared)
      {
        try
        {
          FluxFinanciersServiceReference.FluxFinanciersClient ffc
= new FluxFinanciersServiceReference.FluxFinanciersClient();
          ffc.Open();

          string devise = kvp.Key;
          ReutersData rd = kvp.Value;
          FluxFinanciersServiceReference.ResponseStatutCours statusCours
= ffc.GetCoursDeChange(
rd.Devise,
rd.ContreDevise);

          if (rd.SensDuMarche == "A")
            rd.Bid = statusCours.Value.Bid.Value;

          if (rd.SensDuMarche == "V")
            rd.Ask = statusCours.Value.Ask.Value;

          ffc.Close();

          //
          // Store the data
          //

          ReutersDataCollection.Add(devise, rd);
        }
        catch (Exception ex)
        {
        }
      }
    }
```

Pour préparer les données Reuters, je parcours la grille et stocke en mémoire le sens du marché et les deux données de devises :

```
public void PrepareReutersData(DataGridView dataGridView)
{
  lock (Svc.LockReuters)
  {
    Svc.ReutersDataCollectionPrepared.Clear();

    int count = 0;
    foreach (DataGridViewRow row in dataGridView.Rows)
    {
      object objectDeviseInGrid = dataGridView
.Rows[count]
.Cells[Convert.ToInt32(RowEnum.Devise)].Value;
      string deviseInGrid = Convert.ToString(objectDeviseInGrid);
      object objectContreDeviseInGrid = dataGridView
.Rows[count]
.Cells[Convert.ToInt32(RowEnum.ContreDevise)].Value;
      string contreDeviseInGrid
= Convert.ToString(objectContreDeviseInGrid);
      object objectSensDuMarche
= dataGridView
.Rows[count]
```

```
.Cells[Convert.ToInt32(RowEnum.SensDuMarchéBB)].Value;
            string sensDuMarche = Convert.ToString(objectSensDuMarche);

            if (deviseInGrid == String.Empty)
            {
              break;
            }

            //
            // Store Data
            //
            ReutersData rd = new ReutersData();
            rd.SensDuMarche = sensDuMarche;
            rd.Devise = deviseInGrid;
            rd.ContreDevise = contreDeviseInGrid;
            rd.Ask = 0;
            rd.Bid = 0;

            Svc.ReutersDataCollectionPrepared.Add(deviseInGrid, rd);

            count++;
          }
        }
      }
```

Maintenant que nous avons codé les méthodes d'accès aux données, il faut relier cela au DataGridView... ces méthodes doivent être appelées toutes les 2 secondes. Il est improbable de la faire dans le thread principal de l'application. Nous allons utiliser un thread & un timer.

Acquisition et affichage

Nous allons utiliser un thread pour récupérer les données hétérogènes en arrière-plan et utiliser un timer pour afficher ces informations dans la grille. Le thread principal de l'application est aussi le thread GUI qui possède la grille de message donc il faut faire les affichages dedans. Le thread fait ses appels à SQL Server (méthodes DB.xxx) et l'appel à Reuters (méthode Svc.xxx). Ces méthodes stockent les données en mémoire. Un timer se déclenche toutes les 2 secondes et si le booléen canDisplayData est vrai, il fait les affichages dans la grille DataGridView.

```
public static void ThreadProc(object data)
{
  while (true)
  {
    if (canCloseApp == true)
    {
      // App is closing so we stop the thread !
      break;
    }

    DB.LoadNominaCompta();
    DB.LoadGlobalCompta();
    DB.LoadSoldeAnticipesNominatif();
    DB.LoadNominaAnticipe();
    Svc.CallReutersWS();

    canDisplayData = true;

    Thread.Sleep(2000);
  }
}

public void EnableThread()
{
  Thread thread = new Thread(MainFrame.ThreadProc);
  thread.Start(null);
}


private void _timer_Tick(object sender, EventArgs e)
{
  View view = new View();
  view.PrepareReutersData(dataGridView1);

  if (canDisplayData == true)
  {
    View view2 = new View();
    view2.DisplayNominaCompta(dataGridView1);
    view2.DisplayGlobalCompta(dataGridView1);
    view2.DisplaySoldeAnticipesNominatif(dataGridView1);
    view2.DisplayNominaAnticipe(dataGridView2);
    view2.DisplayReutersData(dataGridView1);

    // Reset the boolean
    canDisplayData = false;
  }
}
```

OK pour le principe. Un thread acquière les données et le timer les affiche. Mais comment se fait l'affichage ? C'est très simple.

```
public void DisplayNominaCompta(DataGridView dataGridView)
{
  int count = 0;
  foreach (NominaCompta nc in DB.NominaComptaCollection)
  {
    dataGridView.Rows[count]
      .Cells[Convert.ToInt32(RowEnum.NominaCompta)]
      .Value = nc.Position;

    dataGridView.Rows[count]
      .Cells[Convert.ToInt32(RowEnum.Devise)]
      .Value = nc.Devise;

    count++;
  }
}
```

Chaque méthode d'affichage est externalisée et prend en paramètre un DataGridView. L'accès aux colonnes se fait via l'énumération RowEnum. A ce stade du développement, les données sont actualisées à intervalle régulier mais c'est toujours de la data en noir et blanc... Et puis, on est censé bypasser Excel et ses formules calculées ; comment faire ?

Les formules calculées

Ça c'est un sujet intéressant ! Les formules doivent être réactives et il faut faire une liaison entre le code C# qui fait le calcul et la cellule du DataGridView. On fait comment ? La solution : on va se greffer sur l'évènement

OnChange de la grille. Et à chaque fois qu'une cellule est modifiée on va vérifier si une colonne entrant dans un ou des calculs a été modifiée. Si c'est le cas, on exécute la formule. Une formule ça ressemble à ça :

```
public class Formulas
{
    public static void Formula_PositionManuelleEnKDevises8(
DataGridView dataGridView, int col, int row)
    {
      if (col == Convert.ToInt32(RowEnum.NominaManuel)
        || col == Convert.ToInt32(RowEnum.GlobalManuel))
      {
        double nm = GridHelper.GetDoubleValue(dataGridView, row,
          RowEnum.NominaManuel);
        double gm = GridHelper.GetDoubleValue(dataGridView, row,
          RowEnum.GlobalManuel);

        double sum = nm + gm;
        GridHelper.SetDoubleValue(dataGridView, row,
          RowEnum.PositionManuelleEnKDevises8, sum);
      }
    }
```

La classe Formulas contient un ensemble de fonctions static qui prennent en paramètre :

- Un DataGridView ;
- Un indice de colonne ;
- Un indice de ligne .

On fait un test pour savoir si l'indice de colonne qui change est impliqué dans une formule. Si oui, on réapplique la formule et on positionne la valeur dans la grille. Voyons à présent comment est pris en charge l'évènement de modification d'une cellule :

```
    public void EnableCellChanged_DGV1(DataGridView dataGridView1)
    {
      dataGridView1.CellValueChanged += DataGridView1_CellValueChanged;
    }

    private void DataGridView1_CellValueChanged(
object sender,
DataGridViewCellEventArgs e)
    {
      DataGridView dataGridView = (DataGridView)sender;
      int col = e.ColumnIndex;
      int row = e.RowIndex;
      object objValue = dataGridView.Rows[row].Cells[col].Value;
      string value = Convert.ToString(objValue);
      string str = String.Format("row={0} col={1} value={2}", row, col, value);

      Formulas.Formula_PositionEnKDevises7Bis(dataGridView, col, row);
      Formulas.Formula_PositionManuelleEnKDevises8(dataGridView, col, row);
      Formulas.Formula_ContrevaleurPositionManuelle12(dataGridView, col, row);
      Formulas.Formula_CalculGainsPertesPotentiels(dataGridView, col, row);
      //
      // After all of these formulas, Reapply Thèmes.
      //
      ApplyColorTheme_DGV1(dataGridView);
      ApplyColorThemeOnValues_DGV1(dataGridView);
    }
```

L'évènement CellValueChanged permet de réappliquer les formules et aussi, à la fin, d'appliquer un thème graphique sur une cellule. Par exemple, la fonction ApplyColorTheme positionne des colonnes en vert et en rose et les devises en gris. La fonction ApplyColorThemeOnValues permet d'écrire en rouge les chiffres à virgule.

```
    public void ApplyColorThemeOnValues_DGV1(DataGridView dataGridView)
    {
      int rowCount = 10;

      for (int c = 0; c < rowCount; c++)
      {
        AddMinusSignOnRowCell(dataGridView, c, RowEnum.SoldeDesAnticipésNominatif);
        AddMinusSignOnRowCell(dataGridView, c, RowEnum.NominaCompta);
        AddMinusSignOnRowCell(dataGridView, c, RowEnum.NominaManuel);
...
        AddMinusSignOnRowCell(dataGridView, c, RowEnum.GainsPertesRealises);
        AddMinusSignOnRowCell(dataGridView, c, RowEnum.StopLoss);
      }
    }

    public void AddMinusSignOnRowCell(DataGridView dataGridView,
int c, RowEnum enumValue)
    {
      DataGridViewCell cell;

      cell = dataGridView.Rows[c].Cells[Convert.ToInt32(enumValue)];
      Object value = cell.Value;
      string strValue = Convert.ToString(value);
      if (String.IsNullOrEmpty(strValue) == true)
        return;

      double dValue;
      if (Double.TryParse(strValue, out dValue) == false)
        return;

      double doubleValue = Convert.ToDouble(value);
      if (doubleValue < 0)
        cell.Style.ForeColor = Color.Red;
      else
        cell.Style.ForeColor = Color.Black;
    }
```

Pour être capable de gérer les données incorrectes ; faute de frappe et autres ; il faut prendre des précautions lorsque l'on manipule les données de la grille. Il est possible de gérer des formats de cellules mais moi j'ai préféré gérer tout à la main. Dans la grille, tout est Object donc il faut gérer soit même les conversions de types.