



***Droits relatif à la diffusion :***

*L'ensemble de cet article ainsi que les codes mis à disposition sont la propriété de leur auteur. La reproduction même partielle de tout ou partie est interdite sans accord écrit préalable de l'auteur.*

## 1. Introduction.

---

Le PropertyGrid est un contrôle méconnu du Framework .NET sûrement dû au fait qu'il ait été oublié de la ToolBox de Visual Studio.

À travers cet article, je vais vous montrer comment utiliser ce contrôle dans vos applications pour en gérer les configurations.

Ce contrôle permet d'afficher et de modifier les propriétés d'un objet, vous le retrouvez dans Visual Studio dans le Designer pour modifier les propriétés des composants que vous disposez dans vos formulaires.

## 2. Ajouter le PropertyGrid dans votre projet.

---

La première chose à faire pour travailler avec le PropertyGrid, c'est de l'ajouter à votre boîte à outils, vous pourrez ensuite le glisser dans n'importe quel formulaire ou `UserControl` de vos applications.

Vous allez donc faire un clic droit sur la boîte à outils de Visual Studio et cliquer sur le menu 'Ajouter / Supprimer des éléments', dans la fenêtre qui s'ouvre, cochez la ligne commençant par 'PropertyGrid' puis cliquez sur le bouton 'OK'.

Vous êtes maintenant prêt à utiliser le PropertyGrid dans vos projets.

## 3. Aborder le PropertyGrid

---

Ce contrôle permet en fait d'éditer les propriétés de n'importe quel objet, c'est pourquoi nous allons voir comment l'utiliser pour gérer la configuration d'une application.

Prenons la classe suivante, elle symbolisera la configuration de notre application :

```
public class ClasseExemple
{
    [Flags]
    public enum CouleurDrapeau {
```

```

        Rouge = 1,
        Vert = 2,
        Bleu = 4,
        Blanc = 8,
        Jaune = 16,
        Noir = 32,
        Marron = 64
    }

    public ClasseExemple() {}

    private string fichier;
    public string Fichier {
        get {return fichier;}
        set {fichier = value;}
    }

    private string folder;
    public string Dossier {
        get {return folder;}
        set {folder = value;}
    }

    private Color color = Color.White;
    public Color Couleur {
        get {return color;}
        set {color = value;}
    }

    private int depth = 50;
    public int Profondeur {
        get {return depth;}
        set {depth = value;}
    }

    private FormStartPosition startPosition =
        FormStartPosition.CenterScreen;
    public FormStartPosition PositionDepart {
        get {return startPosition;}
        set {startPosition = value;}
    }

    private CouleurDrapeau couleurDrapeau =
        CouleurDrapeau.Blanc | CouleurDrapeau.Jaune;
    public CouleurDrapeau Drapeau {
        get {return couleurDrapeau;}
        set {couleurDrapeau = value;}
    }
}

```

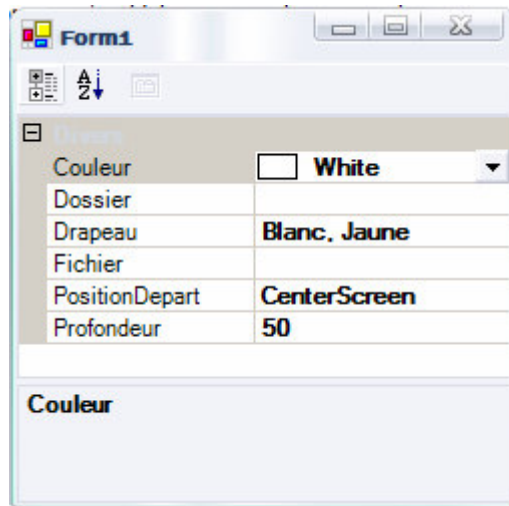
Nous allons instancier cette classe dans notre application et la lier à notre PropertyGrid :

```

ClasseExemple exemple = new ClasseExemple() ;
propertyGrid1.SelectedObject = exemple;

```

Si nous exécutons notre application, nous pouvons voir que notre contrôle affiche bien les différentes propriétés de notre objet et qu'il est possible de les modifier.



Par contre, nos propriétés sont toutes mélangées et je trouve qu'il serait plus intéressant de les regrouper par catégories. Pour cela, nous allons donner un attribut à nos propriétés : `Category("maCategorie")`.

```
public class ClasseExemple
{
    [Flags]
    public enum CouleurDrapeau {
        Rouge = 1,
        Vert = 2,
        Bleu = 4,
        Blanc = 8,
        Jaune = 16,
        Noir = 32,
        Marron = 64
    }

    public ClasseExemple() {}

    private string fichier;
    [Category("Fichier")]
    public string Fichier {
        get {return fichier;}
        set {fichier = value;}
    }

    private string folder;
    [Category("Fichier")]
    public string Dossier {
        get {return folder;}
        set {folder = value;}
    }

    private Color color = Color.White;
    [Category("Couleurs")]
    public Color Couleur {
        get {return color;}
        set {color = value;}
    }

    private int depth = 50;
    public int Profondeur {
```

```

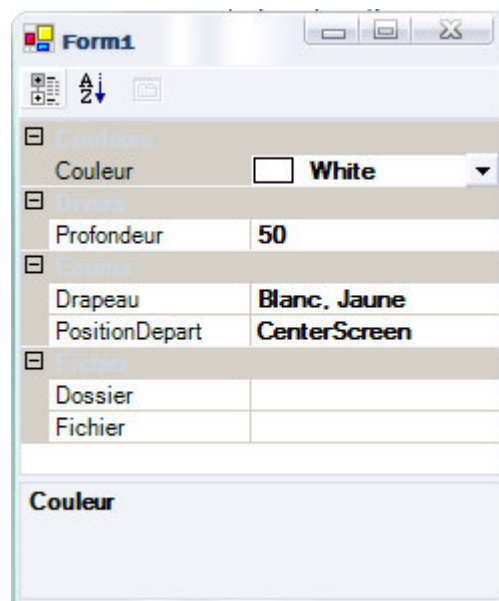
        get {return depth;}
        set {depth = value;}
    }

    private FormStartPosition startPosition =
        FormStartPosition.CenterScreen;
    [Category("Enums")]
    public FormStartPosition PositionDepart {
        get {return startPosition;}
        set {startPosition = value;}
    }

    private CouleurDrapeau couleurDrapeau =
        CouleurDrapeau.Blanc | CouleurDrapeau.Jaune;
    [Category("Enums")]
    public CouleurDrapeau Drapeau {
        get {return couleurDrapeau;}
        set {couleurDrapeau = value;}
    }
}

```

Exécutons à nouveau notre application et constatons le changement. Vous pouvez voir maintenant que nos différentes propriétés sont rangées dans les catégories que nous leur avons assignées.



Vous pouvez voir aussi que le PropertyGrid affiche en gras les valeurs, ce sont en fait les valeurs qui diffèrent de celles par défaut, hors nous n'en avons défini aucune. Nous allons donc de nouveau utiliser un autre attribut : `DefaultValue`.

```

public class ClasseExemple
{
    [Flags]
    public enum CouleurDrapeau {
        Rouge = 1,
        Vert = 2,
        Bleu = 4,
        Blanc = 8,
        Jaune = 16,
        Noir = 32,
        Marron = 64
    }
}

```

```

public ClasseExemple() {}

private string fichier;
[Category("Fichier")]
[DefaultValue("c:\\msdos.sys")]
public string Fichier {
    get {return fichier;}
    set {fichier = value;}
}

private string folder;
[Category("Fichier")]
[DefaultValue("c:\\")]
public string Dossier {
    get {return folder;}
    set {folder = value;}
}

private Color color = Color.White;
[Category("Couleurs")]
[DefaultValue(typeof(Color), "White")]
public Color Couleur {
    get {return color;}
    set {color = value;}
}

private int depth = 50;
[DefaultValue(50)]
public int Profondeur {
    get {return depth;}
    set {depth = value;}
}

private FormStartPosition startPosition =
    FormStartPosition.CenterScreen;
[Category("Enums")]
[DefaultValue(typeof(FormStartPosition), "CenterScreen")]
public FormStartPosition PositionDepart {
    get {return startPosition;}
    set {startPosition = value;}
}

private CouleurDrapeau couleurDrapeau =
    CouleurDrapeau.Blanc | CouleurDrapeau.Jaune;
[Category("Enums")]
[DefaultValue(typeof(CouleurDrapeau), "Blanc, Jaune")]
public CouleurDrapeau Drapeau {
    get {return couleurDrapeau;}
    set {couleurDrapeau = value;}
}
}

```

Nous obtenons ainsi le résultat suivant :

On voit bien sur cette capture qu'il est possible de renseigner facilement les options d'une application... mais ce qui serait intéressant, ce serait d'avoir une description de chaque option ; pour cela, il existe encore un attribut : `Description`, que nous utiliserons comme ceci :

```
[Description("Sélectionnez les couleurs du drapeau")]
[Category("Enums")]
[DefaultValue(typeof(CouleurDrapeau), "Blanc, Jaune")]
public CouleurDrapeau Drapeau {
    get {return couleurDrapeau;}
    set {couleurDrapeau = value;}
}
```

Voilà... nous avons enfin terminé.

## 4. Étendre le PropertyGrid.

Certaines propriétés proposent des assistants pour saisir leur valeur, ce sont la plupart du temps des contrôles flottant permettant le choix rapide entre plusieurs valeurs, couleurs... En


natif, le PropertyGrid offre des éditeurs pour de nombreux types de données (Color, Enum, Font...), mais nous pouvons définir nos propres éditeurs, ce sera le thème de ce paragraphe.

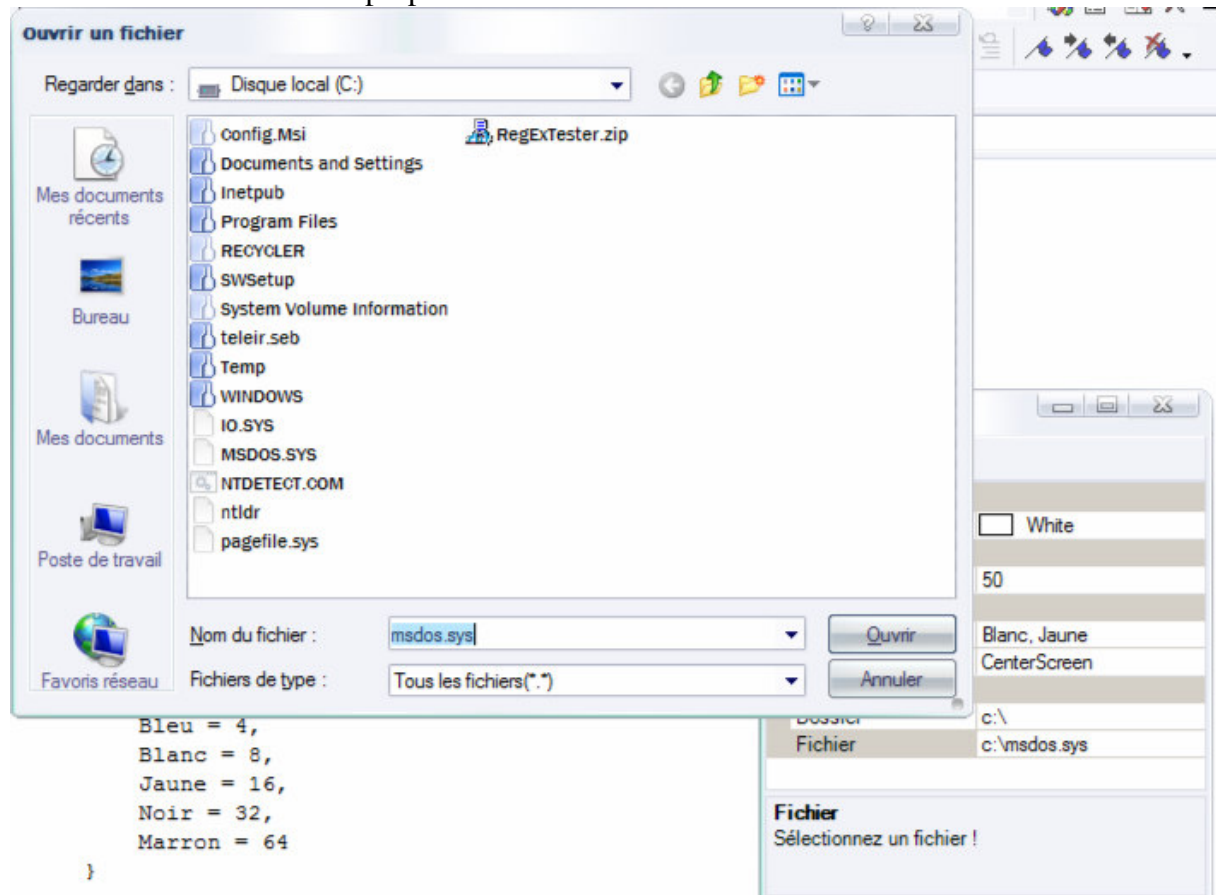
Pour commencer, nous allons découvrir un nouvel attribut : `Editor`, et nous allons proposer pour notre propriété 'Fichier' un dialogue de sélection de fichier à l'aide d'un éditeur standard.

```
[Editor(typeof(FileNameEditor), typeof(UITypeEditor))]
[Description("Sélectionnez un fichier !")]
[DefaultValue("c:\\msdos.sys")]
[Category("Fichier")]
public string Fichier
{
    get {return fichier;}
    set {fichier = value;}
}
```

`FileNameEditor` faisant parti de l'espace de nom `System.Windows.Forms.Design` vous devrez ajouter la référence sur l'assembly `System.Design.dll` et ajouter la directive suivante en haut de votre classe :

```
using System.Windows.Forms.Design;
```

A l'exécution, vous remarquerez l'apparition d'un bouton . Lorsque vous cliquez dessus, une fenêtre s'ouvre et vous propose de choisir un fichier.



Le Framework étant bien fait, il nous offre la possibilité de créer nos propres éditeurs. Nous allons donc en créer 2 qui vont nous permettre de modifier la valeur des propriétés 'Profondeur' et 'Dossier'.

### a. Création d'un éditeur Modal.

#### Définition :

En informatique, on dit qu'une fenêtre est dite 'Modale' lorsqu'elle empêche tout autre fenêtre d'une même application d'être activée. En résumé, une fenêtre modale est la seule fenêtre active d'une application.

Cet éditeur existe déjà, mais pour l'exercice, il est intéressant de voir comment il fonctionne et de le reprogrammer.

Nous allons donc créer un éditeur nous permettant de sélectionner un dossier pour notre propriété 'Dossier'.

À la lecture de la documentation MSDN, nous apprenons dans le paragraphe sur la création d'éditeur personnalisé qu'il faut dériver la classe `System.Drawing.Design.UITypeEditor` et qu'il faut au minimum surcharger les méthodes `GetEditStyle` et `EditValue`.

`GetEditStyle` vous permet de définir le type d'éditeur, il en existe 2 types : `DropDown` et `Modal`. `EditValue` est appelé au moment où vous cliquez sur le bouton ouvrant cet éditeur.

Pour commencer, nous allons préciser que notre éditeur sera modal en surchargeant la méthode `GetEditStyle` :

```
public override UITypeEditorEditStyle GetEditStyle(
    System.ComponentModel.ITypeDescriptorContext context) {

    return UITypeEditorEditStyle.Modal;
}
```

Puis nous allons écrire son comportement lorsque l'utilisateur cliquera sur le bouton d'édition.

```
public override object EditValue(
    System.ComponentModel.ITypeDescriptorContext context,
    IServiceProvider provider,
    object value) {

    using (FolderBrowserDialog fbd =
        new FolderBrowserDialog()) {
        fbd.SelectedPath = (string)value;
        fbd.ShowNewFolderButton = true;
        fbd.RootFolder = Environment.SpecialFolder.Desktop;
        if (fbd.ShowDialog() == DialogResult.OK) {
            value = fbd.SelectedPath;
        }
    }
    return value;
}
```

Ce code n'est pas très compliqué. La méthode ouvre la boîte de dialogue standard « Browse-ForFolder », définit le répertoire par défaut avec la valeur déjà saisie dans le PropertyGrid, accepte la création de nouveaux répertoires et spécifie que le répertoire de base de l'arborescence sera le 'Bureau' de l'utilisateur.

Maintenant que notre éditeur est créé, nous pouvons modifier notre classe pour que le PropertyGrid l'utilise lors de la modification de notre valeur.

Toujours à l'aide de l'attribut Editor, nous spécifions non plus un éditeur standard mais notre éditeur personnalisé :

```
private string folder = "c:\\";
[Editor(typeof(Editors.FolderEditor), typeof(UITypeEditor))]
[Description("Sélectionnez un dossier")]
[DefaultValue("c:\\")]
[Category("Fichier")]
public string Dossier {
    get {return folder;}
    set {folder = value;}
}
```

### ***b. Création d'un éditeur DropDown.***

Les éditeurs de type DropDown permettent de modifier une valeur sans que la fenêtre qui contient le PropertyGrid ne perde le focus, à la manière d'un ComboBox.

Il est possible d'afficher le contrôle de notre choix dans la zone de DropDown, ce qui donne beaucoup d'intérêt à l'utilisation de cette méthode.

Nous allons dans l'exemple suivant afficher un contrôle 'TrackBar' pour modifier une valeur numérique comprise entre 0 et 100.

Nous commençons donc par la surcharge de la méthode GetStyle :

```
public override UITypeEditorEditStyle GetEditStyle(
    ITypeDescriptorContext context) {
    return UITypeEditorEditStyle.DropDown;
}
```

Puis nous traitons l'affichage du DropDown. :

```
public override object EditValue(
    ITypeDescriptorContext context,
    IServiceProvider provider,
    object value) {
    if (value.GetType() != typeof(int) ||
        (int)value < 0 ||
        (int)value > 100) {

        return 0;
    }

    IWindowsFormsEditorService edSvc =
        (IWindowsFormsEditorService)provider.GetService(
            typeof(IWindowsFormsEditorService));
    if( edSvc != null ) {
        this.context = context;
        // Initialise le slider
        System.Windows.Forms.TrackBar trackbar = new TrackBar();
        trackbar.AutoSize = false;
        trackbar.LargeChange = 10;
        trackbar.Maximum = 100;
        trackbar.Minimum = 0;
        trackbar.Orientation = Orientation.Horizontal;
        trackbar.SmallChange = 1;
        trackbar.TickFrequency = 10;
```

```

        trackbar.TickStyle = TickStyle.BottomRight;
        trackbar.Value = (int)value;
        // affichage du Slider dans le dropdown
        edSvc.DropDownControl( trackbar );
        // retourne la valeur sélectionnée
        return trackbar.Value;
    }
    this.context = null;
    return value;
}

```

Nous vérifions dans cet exemple que la valeur contenue dans le PropertyGrid est bien numérique et qu'elle est comprise entre 0 et 100. Vous pouvez remarquer que nous effectuons une tâche supplémentaire par rapport à l'éditeur modal. Nous recherchons cette fois-ci un objet de type `IWindowsFormsEditorService` qui nous permet de lier notre `TrackBar` au `PropertyGrid`.

Une fois la valeur sélectionnée, nous la retournons pour qu'elle soit affectée à la propriété et renseignée dans la grille.

Il nous reste plus qu'à modifier notre classe pour que le `PropertyGrid` considère notre éditeur personnalisé lors de la modification de notre propriété :

```

private int depth = 50;
[Editor(typeof(Editors.SliderEditor), typeof(UITypeEditor))]
[Description("Saisissez la profondeur")]
[DefaultValue(50)]
public int Profondeur {
    get {return depth;}
    set {depth = value;}
}

```

## 5. Conclusion.

---

Nous venons de voir qu'il est simple d'utiliser le `PropertyGrid` dans nos applications et qu'il n'est pas plus difficile de l'étendre de manière à pouvoir gérer nos propres éditeurs. Nous verrons dans un prochain article comment changer le libellé des propriétés et personnaliser l'affichage des valeurs.

En attendant, je vous invite à étudier la source jointe à cet article et plus particulièrement l'éditeur `EnumEditor` qui vous permet de définir la valeur d'une propriété dont le type et un enum avec l'attribut `[flags]`.



L'ensemble ou partie de ce document ainsi que le code mis à disposition, ne peut être diffusé ailleurs sans autorisation préalable

elise.dupont@europe.com - www.dotnet-tech.com - 2004