Programación GUI con Avalonia en C#

Índice

Programación GUI con Avalonia en C#

1 Introducción

La programación GUI (interfaz gráfica de usuario), es vital en aplicaciones de escritorio. En el momento de escribir este documento, aunque existen varias opciones, la más destacable es **Avalonia** por su carácter multiplataforma y su soporte en múltiples entornos. Otras opciones populares son:

- WinForms (*Windows Forms*): Aunque su soporte estaba incluido dentro del propio *runtime* de **mono**, desde el advinimiento de .NET solo es posible desarrollar y ejecutar una aplicación con WinForms en una máquina ejecutando el sistema operativo Windows.
- WPF (*Windows Presentation Framework*): es el GUI por defecto en Visual Studio, pero de manera similar a *WinForms*, no es multiplataforma.

2 Instalación

La mejor forma de comenzar con Avalonia es, sin duda, utilizar las plantillas de nuevo proyecto disponibles para este entorno gráfico. La forma de hacer es utilizar *dotnet new –install*, y el nombre de las plantillas es *avalonia.templates*.

```
$ dotnet new --install avalonia.templates
```

Una vez instaladas las plantillas, que ya definen las referencias necesarias en el proyecto para enlazar con la librería, es posible crear ya un proyecto. Para ello debemos escoger la plantilla *Avalonia .NET Core App*, de nombre corto *avalonia.app*.

```
$ dotnet new avalonia.app -o helloworldavalonia
The template "Avalonia .NET Core App" was created successfully.

$ cd helloworldavalonia

$ ls
App.axaml      App.axaml.cs      helloworldavalonia.csproj      MainWindow.axaml
MainWindow.axaml.cs  Program.cs

$ dotnet run
```

Como se puede ver en las líneas anteriores, el proyecto se crea y se puede ejecutar directamente, mostrando una ventana con el mensaje “Bienvenido a Avalonia” en su interior.

Existen otras plantillas de *Avalonia* que se pueden utilizar, aunque en este texto asumiremos la anterior, que es la más simple. La primera de ellas es *Avalonia .NET Core MVVM App,* con nombre corto *avalonia.mvvm*. Este tipo de proyecto está aconsejado para aquellos usuarios que estén acostumbrados a trabajar con *WPF* en *Windows*, pues sigue los mismos principios. La segunda es *Avalonia Cross Platform Application*, de nombre corto *avalonia.xplat*, pensada para aplicaciones destinadas a móviles y otras plataformas.

3 Paneles

La programación de interfaces gráficas de usuario en Avalonia se realiza mediante controles que se agrupan en paneles (incluso de manera recursiva). Esta agrupación en paneles es lo que hace que la aplicación tenga una apariencia ordenada y limpia, además de permitir que sea insensible a el redimensionado de la ventana.

4 Tipos de paneles

Los tipos de paneles que veremos son: **DockPanel**, y **StackPanel**. Los paneles pueden insertarse unos dentro de otros, de manera recursiva, utilizando sus propiedades *Children*, método *Add()*.

4.1 DockPanel

Es el tipo más básico y probablemente, el más utilizado. Define una serie de áreas en las que colocar controles, y son los propios controles los que, en su propiedad *DockPanel.Dock* (de tipo enumerado *Dock*), indican en que parte del panel van a ser situados (como se ve en la Figura 1).

Figura 1: Botones en un DockPanel.

Las zonas del panel son: *Left*, *Right*, *Top*, y *Down*. El *último* elemento del panel se sitúa en el centro y se redimensiona hasta ocupar todo el área libre de la ventana con respecto al resto de controles ya ubicados a la izquierda (*left*), derecha (*right*), arriba (*top*) y abajo (*bottom*). Asi, las cuatro primeras se adaptan al tamaño de los controles que albergan, y ya no se modifican, mientras que la última es la que recibe el nuevo tamaño cuando la ventana es redimensionada.
Hay que resaltar que, a la hora de añadir los componentes al panel, el último siempre debe ser el que va a ocupar el mayor espacio, tras el resto.

Ejemplo (solo código)

```
var pnlNombre = new DockPanel();

var lblNombre = new Label();
lblNombre.Content = “Nombre:”;
DockPanel.SetDock( lblNombre, Dock.Left );

var edNombre = new TextBox();

pnlNombre.Children.Add( lblNombre );
pnlNombre.Children.Add( edNombre );
```

Ejemplo (con XAML)

```
<DockPanel>
      <Label DockPanel.Dock=”Left” Name=”lblNombre” Content=”Nombre:” />
      <TextBox Name=”EdNombre” />
</DockPanel>
```

4.2 StackPanel

En este panel los controles se disponen redimensionándolos para que normalmente (a no ser que los controles tengan restricciones de tamaño), ocupen tamaños iguales, ocupando todo el espacio del control.

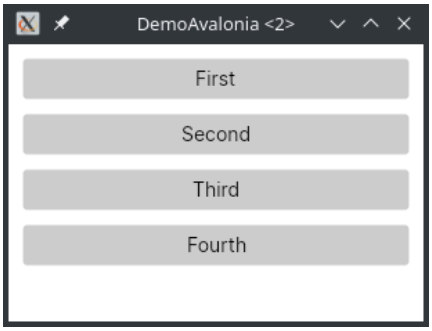

Figura 2: Botones en un StackPanel.

Los **StackPanel** pueden añadir los controles siguiendo hasta básicamente dos patrones que se configuran en su propiedad *Orientation*: *Vertical* (de arriba a abajo), y *Horizontal* (de izquierda a derecha). Por defecto, los controles se añaden de abajo a arriba.

Ejemplo (solo código)

```
var pnlBotonera = new StackPanel();

var btInsertar = new Button();
btInsertar.Content = “+”;

var btEliminar = new Button();
btEliminar.Content = “-”;

pnlBotonera.Children.Add( btInsertar );
pnlBotonera.Children.Add( btEliminar );
```

Ejemplo (XAML)

```
<StackPanel>
      <Button Name=”BtInsertar” Content=”+” />
      <Button Name=”BtEliminar” Content=”-”/>
</StackPanel>
```

5 Controles básicos

Los siguientes son controles básicos, que se emplean principalmente para recoger la entrada del usuario. Un atributo común a estos controles es *Name*. Dándole un identificador a un control dentro de este atributo permite que sea posible referenciar ese control desde el código en C#.

Nótese que por cada ventana *v1* tendremos los archivos *v1.xaml* y *v1.xaml.cs*, de manera que el primero guarda la estructura XAML para describir la interfaz de usuario, y el segundo será el que realmente realizará el manejo de los controles gráficos a través del nombre que le hayamos asignado con el atributo *Name*.

Tabla 1: Atributos comunes de controles.

Atributo	Significado
HorizontalAlignment: enum	Alineación horizontal del control.
HorizontalContentAlignment: enum	Alineación horizontal del contenido del control.
VerticalAlignment: enum	Alineación vertical del control.
VerticalContentAlignment: enum	Alineación vertical del contenido del control.
IsEnabled: bool	Si el control admite interacción. True por defecto.
Margin	Margen alrededor del control.
Padding	Margen desde el borde del control hasta el contenido.
FontSize	Tamaño de la fuente en el control.

5.1 Label

El control *label* se emplea para mostrar texto en una ventana. Por ejemplo, como etiqueta de un campo de texto (ver más adelante).

Ejemplo (XAML)

```
<Label Content="Op1" />
```

Ejemplo (Solo código)

```
var lbl = new Label { Content = “Op1” };
```

5.2 TextBox

El control *textbox* es capaz de guardar una cadena de caracteres introducida por el teclado. Para definirlo como XAML, se utiliza la etiqueta *Textbox*:

Ejemplo (XAML)

```
<TextBox Name="EdOp1" Text="0" TextAlignment="Right" />
```

Ejemplo (solo código)

```
var ed = new TextBox { Text = “0”, TextAlignment = “Right” };
```

Tabla 2: Otros atributos de TextBox.

Atributo	Significado
IsReadOnly: bool	Si el contenido se puede editar. False por defecto.
TextAlignment: TextAlignment enum	TextAlignment.Left por defecto. Puede ser *center* o *right*.
IsEnabled: bool	Si el control admite interacción. True por defecto.

5.3 Button

El control *button* permite capturar la decisión del usuario de comenzar una acción.

Ejemplo (XAML)

```
<Button Name="BtOk" Content="Ok" />
```

Ejemplo (solo código)

```
var btOk = new Button { Content = “Ok” };
```

6 Menús

El menú principal de una ventana se diseña utilizando las etiquetas *Menu* y *MenuItem*. La etiqueta *Separator* permite crear un separador entre opciones. De nuevo, el atributo *Name* permite asignar un identificador para manejar el control desde el propio código.

Para asignar el texto que se mostrará en la opción, se emplea el atributo Header. El subrayado delante de una de las letras del texto de la opción indica el atajo de teclado que servirá para activar la opción sin utilizar el ratón.

Ejemplo (XAML)

```
        <Menu>
            <MenuItem Header="_File">
                <MenuItem Header="_Exit" Name="OpExit" />
            </MenuItem>
            <MenuItem Header="_View">
                <MenuItem Header="_Simple panel" Name="OpSimplePanel" />
                <MenuItem Header="S_tack panel" Name="OpStackPanel" />
                <MenuItem Header="_Form" Name="OpForm" />
                <Separator />
                <MenuItem Header="Demo _gráficas" Name="OpChart" />
                <MenuItem Header="_Message box" Name="OpMessageBox" />
            </MenuItem>
            <MenuItem Header="_Help">
                <MenuItem Header="_About" Name="OpAbout" />
            </MenuItem>
        </Menu>
```

Ejemplo (solo código)

```
        var menu = new Menu();
        var opOpen = new MenuItem{ Header = "_Open" };
        var opClose = new MenuItem{ Header = "_Close" };
        var mFile = new MenuItem{ Header = "_File",
            Items = new[] {
                opOpen, opClose
            }
        };
```

7 DataGrid

El control **DataGrid** se utiliza para mostrar datos en formato de tabla. Funciona de una manera ligeramente distinta a otros controles similares en diferentes *toolkits* gráficos, ya que en este caso refleja una colección, siempre y cuando dicha colección se observable.

El primer paso consiste en añadir los estilos al archivo App.axml, de manera que dentro de <*styles*>, incorporamos la línea:

```
<StyleInclude
       Source="avares://Avalonia.Controls.DataGrid/Themes/Fluent.xaml"/>
```

Sin la línea más arriba, no se mostrará absolutamente nada en el **DataGrid**, como si no existiera.

El siguiente paso es incorporar el **DataGrid** en el XAML de nuestra ventana. Supongamos la siguiente clase:

```
public class Persona {
      public string Nombre { get; set; }
      public string Email { get; set; }
}
```

En un caso tan sencillo, simplemente podemos incorporar el *DataGrid* como sigue:

```
<DataGrid   Name=”DtPersonas” AutoGenerateColumns=”True”>
</DataGrid>
```

Pero es necesario que exista alguna lista de personas para poder incorporarlas al **DataGrid**:

```
public class RegistroPersonas: ObservableCollection<Persona> {
}
```

En MainWindow.axml.cs, creamos un nuevo registro de personas y lo enlazamos con el **DataGrid**:

```
this.registro = new RegistroPersonas();
var dtPersonas = this.FindControl<DataGrid>( “DtPersonas” );

dtPersonas.Items = this.registro;
```

Con esto aparecerá la tabla vacía a excepción de las cabeceras de las columnas. Si creamos diálogos para insertar nuevas personas en el *registro*, estas aparecerán automáticamente.

Concretamente, aparecerán dos: “Nombre” e “Email”. Si queremos personalizar de alguna forma estas columnas, deberemos modificar el XAML de la manera siguiente:

```
<DataGrid   Name=”DtPersonas” AutoGenerateColumns=”False”>
      <DataGrid.Columns>
            <DataGridTextColumn
                        Header=”Nombre”
                        Binding=”{Binding Nombre}”/>
            <DataGridTextColumn
                        Header=”E.mail”
                        Binding=”{Binding Email}”/>

      </DataGrid.Columns>
</DataGrid>
```

8 Eventos

Para gestionar las reacciones a las acciones del usuario, se emplean eventos, a los que se puede responder utilizando *lambdas*. Con diferencia, el evento más utilizado es *Click* (en realidad se dispara cuando se activa el control, no solo cuando se pulsa el botón del ratón, sino también con el teclado).

Aunque en versiones anteriores era necesario utilizar **Control**.*FindControl<T>(id)* para obtener una referencia al control (siendo *id* el identificador dado a *Name*, y *T* el tipo de control, por ejemplo, *Button*), ahora ya se puede directamente utilizar el identificador como un atributo más.

En cuanto al soporte para eventos de C#, este se expresa con el operador +=, de forma que el nuevo gestor de eventos, en lugar de ser el único gestor de eventos disponible, se añade al final de una lista de gestores para ese evento. Además, el evento determina que el gestor de eventos debe aceptar unos parámetros (normalmente, dos), que normalmente se corresponden con el control en el que se ha disparado el evento, y los parámetros para ese evento, respectivamente. Muchas veces no son necesarios estos datos. Finalmente, las lambdas aceptan el subrayado como nombre de parámetro que por un lado tiene que estar, y por el otro no va a ser utilizado.

Un ejemplo para manejar la activación de una opción del menú, y un botón sería el siguiente. Se utiliza *FindControl<>()* aunque como ya se ha dicho, es opcional actualmente. El objetivo es que al pulsar en un botón (*Button*), un texto en la ventana (*Label*) cambie su mensaje a “¡Hola!”, mientras que si se pulsa en un opción del menú *OpQuit* se cierra la ventana y si esta es la ventana principal, se sale de la aplicación.

Ejemplo

```
var btHola = this.FindControl<Button>( "BtHola" );
var lblMensaje = this.FindControl<Label>( "LblMensaje" );
var opQuit = this.FindControl<MenuItem>( "OpQuit" );
btHola.Click += (_, _) => lblMensaje.Text = “¡Hola!”;
opQuit.Click += (_, _) => this.Close();
```

9 Referencias

1. *Avalonia*: http://avaloniaui.net

2. Documentación de *Avalonia*: https://docs.avaloniaui.net/docs/getting-started