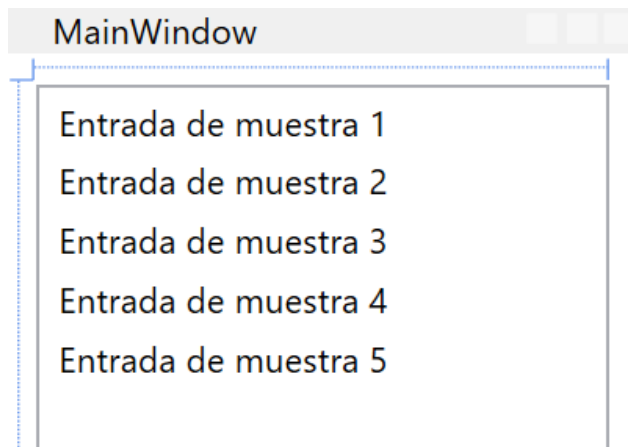


Ejemplo de controles dinámicos

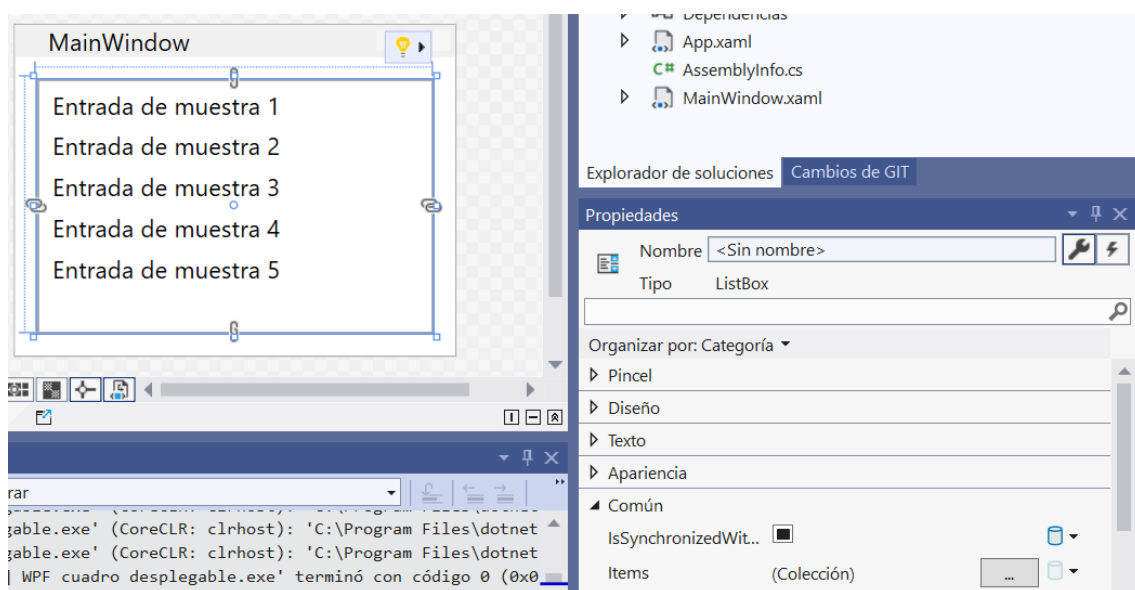
1. ListBox	1
2. ComboBox	6
3. ListView	7
4. DataGrid	10
5. TreeView	14

1. ListBox

Además de presentar los ítems en forma de lista permite seleccionarlos. Por defecto se añaden unos ítems de muestra que aparecen solo en el diseño.

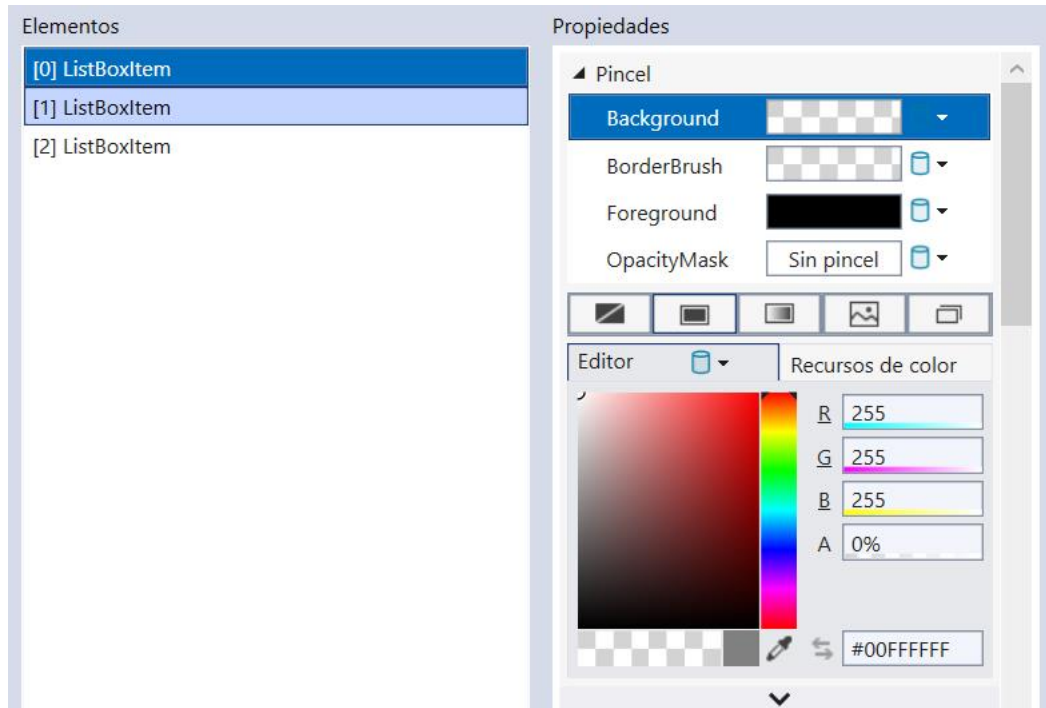


Podemos eliminar estos ítems desde el XAML y añadir los ítems directamente desde la barra de herramientas.

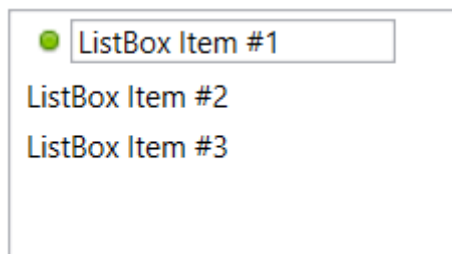


WPF - 6. Ejemplo controles dinámicos

En esta opción podemos añadir ítems de varios tipos. El ítem por defecto para un **ListBox** es **ListBoxItem**. Por cada ítem podemos configurar una serie de estilos y propiedades.



Dentro cada ítem podemos arrastrar desde la herramienta gráfica un control más complejo. Por ejemplo, en este caso añadimos un **StackPanel** con una imagen y un **TextBox**.

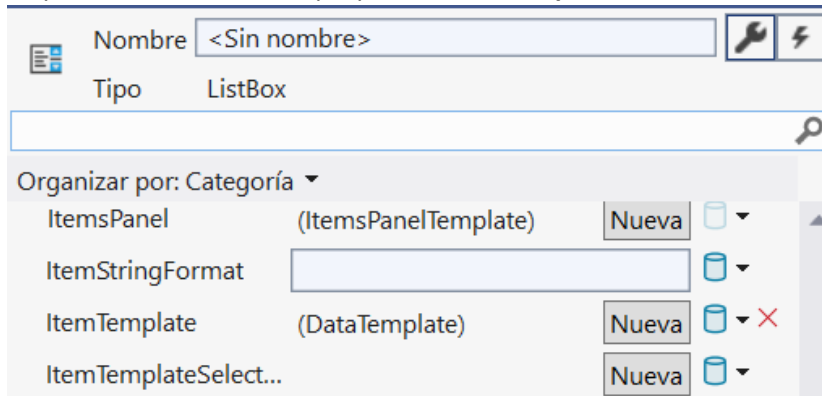


Este sería el código XAML

```
<ListBox>
  <ListBoxItem>
    <StackPanel Orientation="Horizontal">
      <Image Source="/icon.png" />
      <TextBox TextWrapping="Wrap" Text="ListBox Item #1" Width="120" />
    </StackPanel>
  </ListBoxItem>
  <ListBoxItem>ListBox Item #2</ListBoxItem>
  <ListBoxItem>ListBox Item #3</ListBoxItem>
</ListBox>
```

También es posible definir un **DataTemplate** para que todos los datos tengan una plantilla diferente a un String, que es el ítem por defecto.

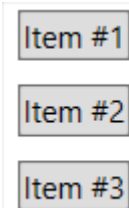
Se puede definir desde la propiedad **ItemTemplate**.



Aunque luego a la hora de añadir los datos para la visualización es más sencillo editar manualmente el documento XAML. Por ejemplo, debajo tenemos una plantilla que muestra todas las filas como un botón.

```
<ListBox Grid.Row="1">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Button Content="{Binding}" Margin="0,0,5,5" />
    </DataTemplate>
  </ListBox.ItemTemplate>
  <system:String>Item #1</system:String>
  <system:String>Item #2</system:String>
  <system:String>Item #3</system:String>
</ListBox>
```

Con la etiqueta "Binding", cada String definido debajo se representará de la siguiente manera.



Pero hay otra forma de automatizar este proceso a través de la propiedad **ItemsSource** y así no tenemos que definir manualmente cada ítem.

Lo que hace **ItemsSource** es asociar una variable (en este caso de tipo lista) con los ítems del **ListBox**. Una opción es definirlo directamente desde el código C#. El nombre lbTodoList es simplemente aquel que se define en el XAML para el ListBox.

```

namespace WPF_cuadro_desplegable
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            List<TodoItem> items = new List<TodoItem>();
            items.Add(new TodoItem() { Title = "WPF", Completion = 45 });
            items.Add(new TodoItem() { Title = "C#", Completion = 80 });
            items.Add(new TodoItem() { Title = "Java", Completion = 0 });
            this.lbTodoList.ItemsSource = items;
        }

        public class TodoItem
        {
            public string Title { get; set; }
            public int Completion { get; set; }
        }
    }
}

```

Fíjate que se ha creado una lista de un tipo de datos personalizado que creamos en una clase justo debajo, en el mismo fichero. Estamos creando una lista con una clase, cuyas propiedades se asociarán con **Binding** en el XAML de debajo.


```


<ListBox Grid.Row="2" Name="lbTodoList">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Grid Margin="0,2">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="100" />
                </Grid.ColumnDefinitions>
                <TextBlock Text="{Binding Title}" />
                <ProgressBar Grid.Column="1" Minimum="0" Maximum="100"
Value="{Binding Completion}" />
            </Grid>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>

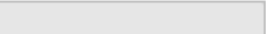
```

Como puedes observar, dentro del **DataTemplate** hemos creado un Grid desde cero con un TextBlock y ProgressBar. Cada uno tiene en la propiedad **Value** el atributo Binding con la propiedad de la clase **TodoItem** correspondiente.

Se visualiza de la siguiente manera.

WPF 

C# 

Java 

Se puede hacer el código todavía más eficiente y reutilizable. Para ello podemos crear los datos como un recurso de WPF. Aunque se puede hacer mediante VisualStudio, la manera más sencilla es editando directamente el archivo XAML.

Supongamos la clase `TodoItem` que creamos anteriormente. Vamos a sobrescribir el método `ToString`

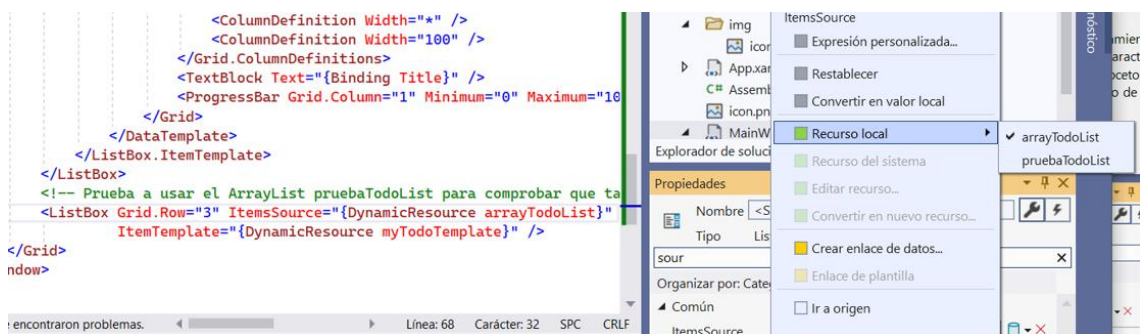
```
public class TodoItem
{
    public string Title { get; set; }
    public int Completion { get; set; }

    // Sobrescribimos el método ToString
    public override string ToString()
    {
        return this.Title + "-" + this.Completion;
    }
}
```

Ahora creamos desde el XAML con el diseño de la página un array de este tipo de la siguiente manera.

```
<Window.Resources>
    <x:Array x:Key="arrayTodoList" Type="{x:Type local:TodoItem}">
        <local:TodoItem Title="WPF" Completion="45"/>
        <local:TodoItem Title="C#" Completion="80"/>
        <local:TodoItem Title="Java" Completion="0"/>
    </x:Array>
</Window.Resources>
```

Desde la propiedad `ItemsSource` asociamos este Array con los ítems a través de un “Recurso local”.



Si no sobrescribiéramos el método `ToString`, se mostraría una referencia al objeto que C# no sabe representar como un string a no ser que lo indiquemos.

```
WPF_cuadro_desplegable.TodoItem
WPF_cuadro_desplegable.TodoItem
WPF_cuadro_desplegable.TodoItem
```

Además de **ItemsSource**, también podemos crear un template similar al anterior como recurso.

```
<DataTemplate x:Key="myTodoTemplate">
    <Label Content="{Binding}" Foreground="#FFB51B1B" />
</DataTemplate>
```

Lo añadimos a **ItemTemplate** de manera similar a como hemos hecho con **ItemSource**. En la captura anterior se puede observar. Al añadir directamente **Binding** en el Content del Label, se corresponde con la representación de ToString cuyo método hemos sobrecargado.

La representación final es la siguiente:

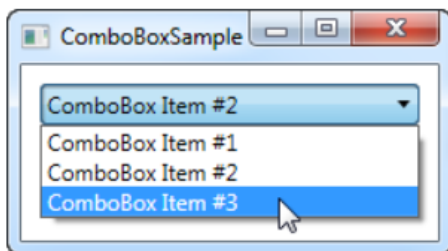
WPF-45

C#-80

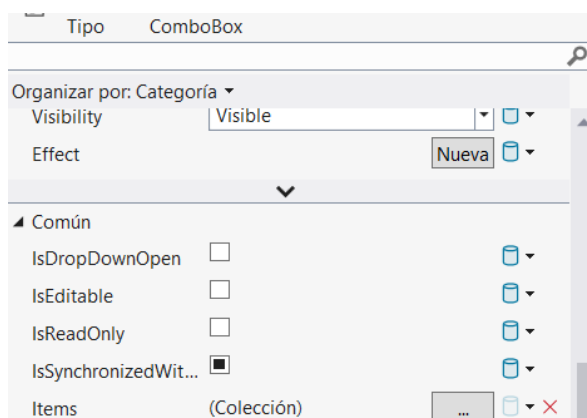
Java-0

2. ComboBox

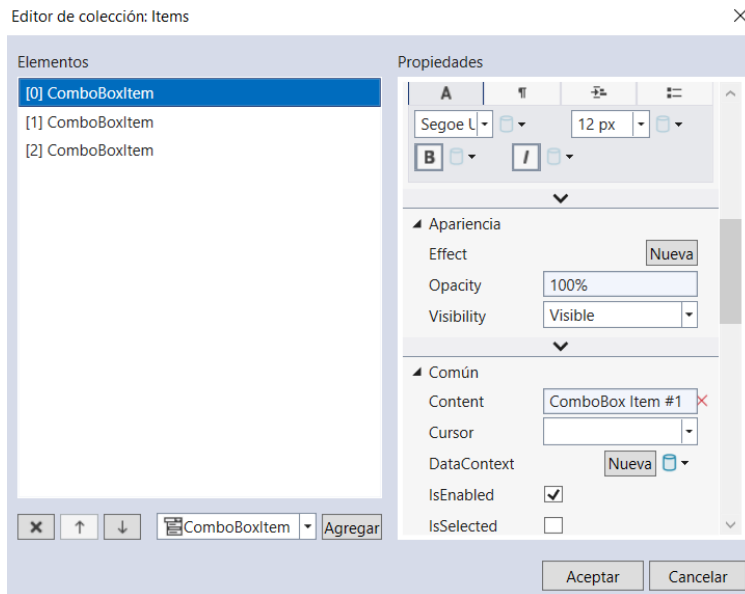
El control **ComboBox** es parecido al control **ListBox** en muchos sentidos, pero usa mucho menos espacio, ya que la lista de ítems se encuentra oculta cuando no se necesita y solo se puede seleccionar un elemento al mismo tiempo.



Para añadir un **ComboBox** basta con arrastrarlo en el diseño. La forma más sencilla de agregar ítems sería desde la barra de herramientas en el apartado **Items**.

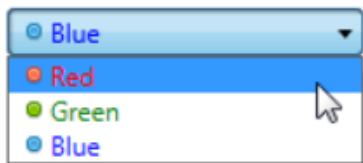


Desde esta sección agregaremos los diferentes elementos. El tipo por defecto es **ComboBoxItem**, aunque puede haber elementos de otros tipos como botones o **CheckBox**.

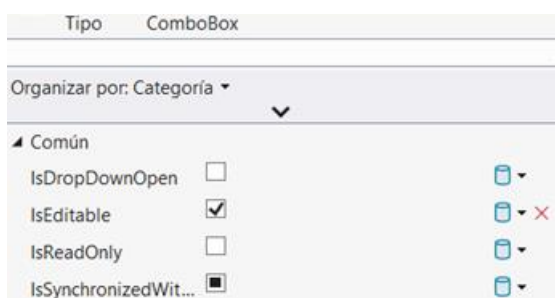


En el menú anterior, podemos cambiar el contenido e indicar si está seleccionado con la propiedad **IsSelected**.

Al igual que en **ListBox** podemos hacer que los ítems tengan contenido personalizado de una manera similar a como se explicaba en la sección correspondiente a este control.



Por otro lado, los ítems de un **ComboBox** pueden ser editables mediante la propiedad específica de este control **IsEditable**.



3. ListView

En su forma más sencilla un **ListView** no presenta grandes diferencias respecto a un **ListBox**. Además también permite **ItemTemplate** para personalizar el contenido.

La potencia de **ListView** radica en las vistas y WPF viene con una vista especializada: la **GridView**. Esto permite organizar la información en columnas. La desventaja es que se debe definir directamente en el XAML y no es posible desde el editor gráfico.

```

<ListView Margin="10" Name="lvUsers">
  <ListView.View ItemsSource="{StaticResource listadoUsuario}">
    <GridView>
      <GridViewColumn Header="Name" Width="120"
        DisplayMemberBinding="{Binding Name}" />
      <GridViewColumn Header="Age" Width="50"
        DisplayMemberBinding="{Binding Age}" />
      <GridViewColumn Header="Mail" Width="150"
        DisplayMemberBinding="{Binding Mail}" />
    </GridView>
  </ListView.View>
</ListView>

```

Dentro de **GridView** hemos definido tres columnas, una para cada uno de los tipos de piezas de datos que deseamos mostrar. La propiedad **Header** se usa para especificar el texto que nos gustaría mostrar para la columna y luego usamos la propiedad **DisplayMemberBinding** para vincular el valor a una propiedad de nuestra clase de usuario.

Para ello necesitamos una clase que contenga las propiedades Name, Age y Mail.

```

public class User
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Mail { get; set; }
}

```

En el código XAML de arriba podemos ver definida la propiedad **ItemsSource** que nos permite acceder a un recurso, que en este caso se ha creado a nivel de ventana y que inicializa una lista con varios datos de prueba.

```

<Window.Resources>
  <Collections:ArrayList x:Key="listadoUsuario">
    <local:User Name="John Doe" Age="42" Mail="john@doe-family.com" />
    <local:User Name="Jane Doe" Age="39" Mail="jane@doe-family.com" />
    <local:User Name="Sammy Doe" Age="7" Mail="sammy.doe@gmail.com" />
  </Collections:ArrayList>
</Window.Resources>

```

El resultado es el siguiente:

Name	Age	Mail
John Doe	42	john@doe-family.com
Jane Doe	39	jane@doe-family.com
Sammy Doe	7	sammy.doe@gmail.com

También es posible cambiar los estilos como en cualquier otro control de WPF. Por ejemplo, vamos a cambiar la alineación de todas las cabeceras de columnas a nivel de ventana (afectará a todos los **GridViewColumn**).

```

<Window.Resources>
  <Style TargetType="{x:Type GridViewColumnHeader}">
    <Setter Property="HorizontalContentAlignment" Value="Right" />
  </Style>
</Window.Resources>

```


Como puedes observar ahora se visualizan a la derecha.

Name	Age	Mail
John Doe	42	john@doe-family.com
Jane Doe	39	jane@doe-family.com
Sammy Doe	7	sammy.doe@gmail.com

También existe la posibilidad de ordenar los ítems del **ListView**, aunque solo se puede desde el código C# de la siguiente manera. Se puede ordenar por tantos campos como queramos.

```
CollectionView view = (CollectionView)
CollectionViewSource.DefaultView(lvUsers.ItemsSource);

view.SortDescriptions.Add(new SortDescription("Age",
ListSortDirection.Ascending));
```

En realidad obtenemos una referencia a la vista y le añadimos un criterio de ordenación.

Por último, vamos a probar a definir un filtro para el listado. Esto se puede hacer con el mismo objeto de la clase **CollectionView** que se usa para ordenar. La desventaja es que tampoco se podría desde el código XAML.

```
CollectionView view = (CollectionView)
CollectionViewSource.DefaultView(lvUsers.ItemsSource);

view.Filter = UserFilter;
```

El filtro que se asigna a la clase de **CollectionView** se define justo debajo con el siguiente método. Este método devuelve true cuando la columna **Name** contiene algún carácter del filtro.

```
private bool UserFilter(object item)
{
    if (String.IsNullOrEmpty(this.txtFilter.Text))
        return true;
    else
        return ((item as User).Name.IndexOf(this.txtFilter.Text,
StringComparison.OrdinalIgnoreCase) >= 0);
}
```

Además, en el evento **TextChanged** cuando se modifica el **TextBox**, si refrescamos el estado de la vista se hará con el filtro.

```
private void txtFilter_TextChanged(object sender, TextChangedEventArgs e)
{
    CollectionViewSource.DefaultView(lvUsers.ItemsSource).Refresh();
}
```

Por tanto, cualquier valor del **TextBox** filtrará la lista según el campo **Name**.

Ja

Name	Age	Mail
Jane Doe	39	jane@doe-family.com

Más información sobre filtros:

<https://docs.microsoft.com/es-es/dotnet/desktop/wpf/data/how-to-filter-data-in-a-view?view=netframeworkdesktop-4.8&viewFallbackFrom=netdesktop-6.0>

4. DataGrid

El control **DataGrid** es parecido al **ListView** cuando usamos un **GridView**, pero ofrece una funcionalidad adicional. Por ejemplo, el **DataGrid** puede generar columnas automáticamente dependiendo de los datos que se le proporciona.

Vamos a probar a crear un **DataGrid** simple para comprobar que las columnas se generan a partir de los datos.

Por defecto nos genera una plantilla que podemos borrar sobrescribiendo la propiedad **ItemsSource**

MainWindow				
SampleInt	SampleStringA	SampleStringB	SampleBool	
1	Cadena de ejemplo A: 1	Cadena de ejemplo B: 1	<input checked="" type="checkbox"/>	
2	Cadena de ejemplo A: 2	Cadena de ejemplo B: 2	<input checked="" type="checkbox"/>	
3	Cadena de ejemplo A: 3	Cadena de ejemplo B: 3	<input checked="" type="checkbox"/>	
4	Cadena de ejemplo A: 4	Cadena de ejemplo B: 4	<input checked="" type="checkbox"/>	
5	Cadena de ejemplo A: 5	Cadena de ejemplo B: 5	<input checked="" type="checkbox"/>	

Basta con crear una clase que representa los datos y añadir el listado y **DataGrid** funcionaría. En la clase de C# con la ventana añadiremos la clase **User** con las filas y columnas del **DataGrid**. Además, añadimos los datos desde el constructor de la ventana **MainWindow** habiendo asignado previamente el nombre de control "dgSimple".

```

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();


        List<User> users = new List<User>();
        users.Add(new User() { Id = 1, Name = "John Doe", Birthday = new
            DateTime(1971, 7, 23) });
        users.Add(new User() { Id = 2, Name = "Jane Doe", Birthday = new
            DateTime(1974, 1, 17) });
        users.Add(new User() { Id = 3, Name = "Sammy Doe", Birthday = new
            DateTime(1991, 9, 2) });

        this.dgSimple.ItemsSource = users;
    }
}

public class User{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime Birthday { get; set; }
}

```

Si ejecutamos la aplicación nos habrá generado la tabla con la información y el formato requeridos.

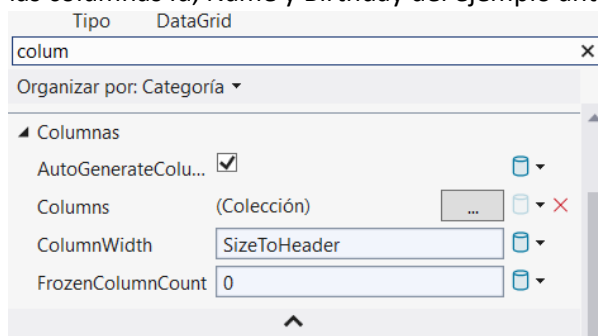
 MainWindow

Id	Name	Birthday
1	John Doe	7/23/1971 12:00:00 AM
2	Jane Doe	1/17/1974 12:00:00 AM
3	Sammy Doe	9/2/1991 12:00:00 AM

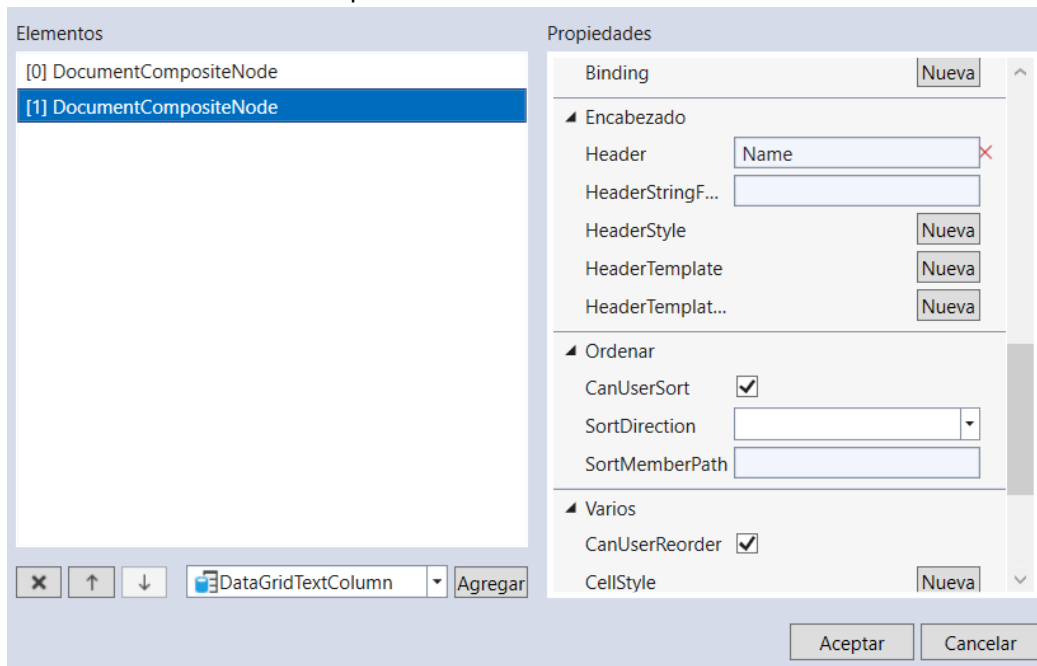
En la aplicación anterior puedes observar que los datos son editables por defecto y las columnas se pueden ordenar.

En algunas situaciones, es posible que queramos definir manualmente las columnas que se muestran, ya sea porque no deseamos todas las propiedades o columnas de la fuente de datos, o simplemente para tener mayor control.

Para ello, en el cuadro de herramientas tenemos que desactivar **AutoGenerateColumns** y añadir las columnas Id, Name y Birthday del ejemplo anterior desde el apartado **Columns**.



Tenemos varios tipos de columna. De momento veremos solo el tipo **DataGridTextColumn**. El nombre se añade con el campo **Header**.



Tras añadir las tres columnas manualmente, vamos a asociar el mismo **ItemsSource** del listado anterior.

```
public MainWindow()
{
    InitializeComponent();

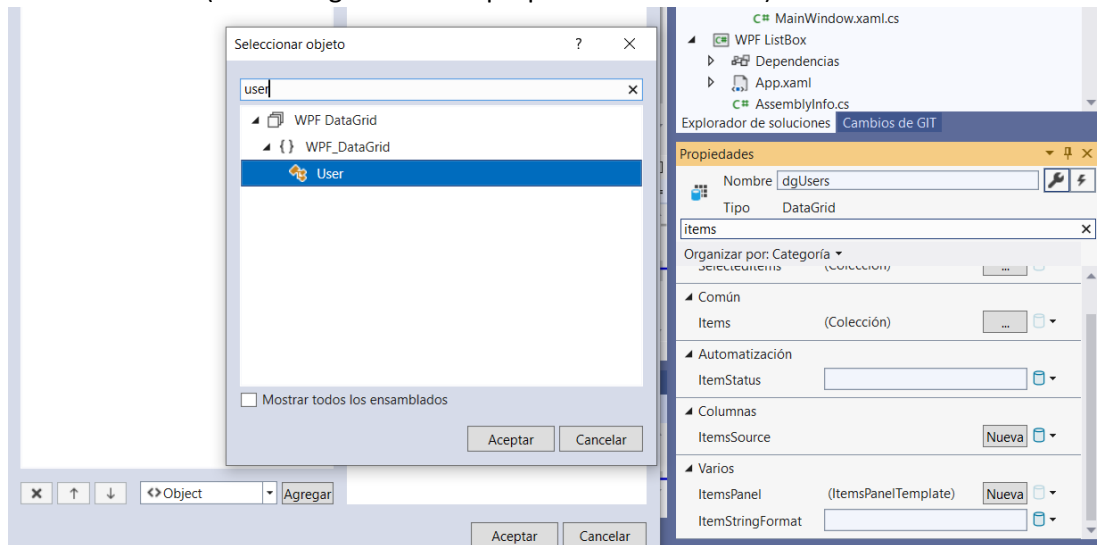
    List<User> users = new List<User>();
    users.Add(new User() { Id = 1, Name = "John Doe", Birthday = new
        DateTime(1971, 7, 23) });
    users.Add(new User() { Id = 2, Name = "Jane Doe", Birthday = new
        DateTime(1974, 1, 17) });
    users.Add(new User() { Id = 3, Name = "Sammy Doe", Birthday = new
        DateTime(1991, 9, 2) });

    this.dgUsers.ItemsSource = users;
}
```

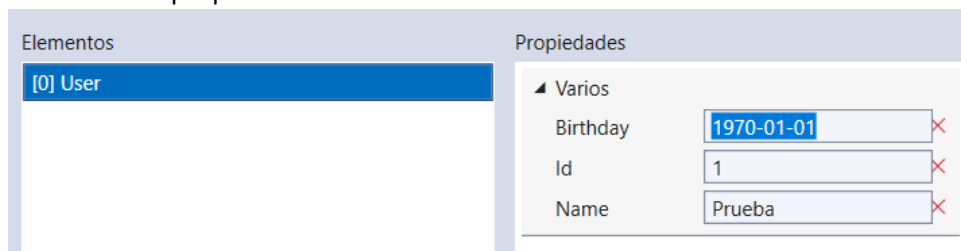
En este caso, tenemos que indicar en el archivo XAML a qué propiedad de la clase **User** hace referencia cada columna con **Binding**.

```
<DataGrid x:Name="dgUsers" Grid.Row="1" AutoGenerateColumns="False">
    <DataGrid.Columns>
        <DataGridTextColumn Header="Id" Binding="{Binding Id}"/>
        <DataGridTextColumn Header="Name" Binding="{Binding Name}"/>
        <DataGridTextColumn Header="Birthday" Binding="{Binding
    Birthday}"/>
    </DataGrid.Columns>
</DataGrid>
```

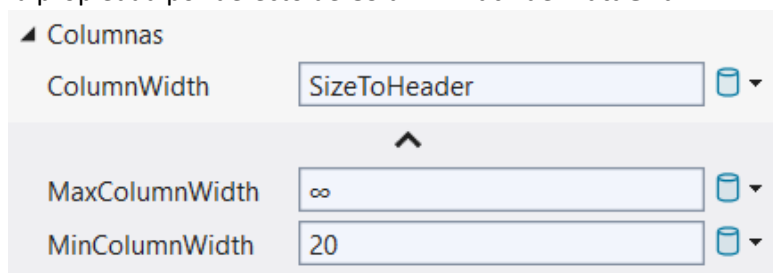
El **Binding** siempre va a ser necesario cuando creamos las columnas manualmente, ya que tenemos que decirle a qué propiedad se refiere. No obstante, en los dos casos anteriores también podemos crear el listado desde la propiedad **Items** para simplificar y ahorrar código en el archivo de C# (donde asignábamos la propiedad **ItemsSource**)



Desde esta opción agregamos un **Object** y buscamos el tipo de datos **User** creado anteriormente como se puede observar en la imagen anterior. Una vez seleccionado, ya nos permite añadir datos con las propiedades de **User**.



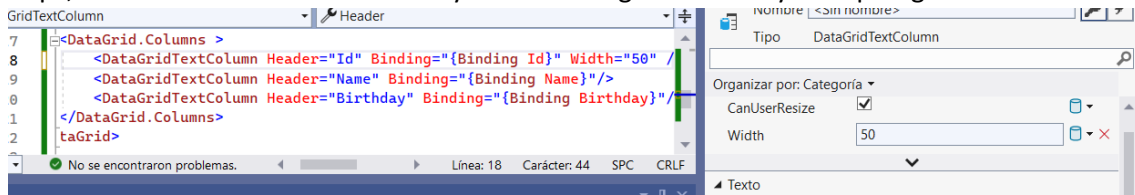
Por otro lado, en los ejemplos anteriores las columnas se ajustan al contenido de las celdas con la propiedad por defecto de **ColumnWidth** del **DataGrid**.



Si queremos que las columnas se adapten al ancho disponible, lo más rápido es cambiar la propiedad **ColumnWidth** al valor ***** (es la unidad de medida que reparte el espacio disponible como tal). Es espacio se reparte equitativamente:

Id	Name	Birthday
1	John Doe	7/23/1971 12:00:00 AM
2	Jane Doe	1/17/1974 12:00:00 AM
3	Sammy Doe	9/2/1991 12:00:00 AM

Hay columnas como Id que no necesitan tanto espacio. Podemos ajustar la proporción modificando el ancho individual de cada columna. Por ejemplo, si queremos que el Id ocupe solo 50px, se sobrescribirá a este tamaño y el resto se siguen distribuyendo por igual.

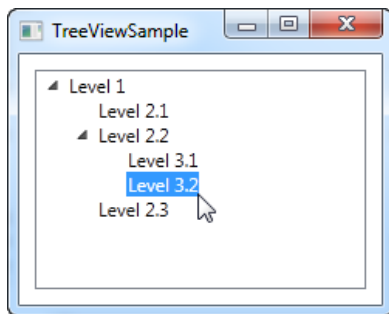


Este sería el resultado:

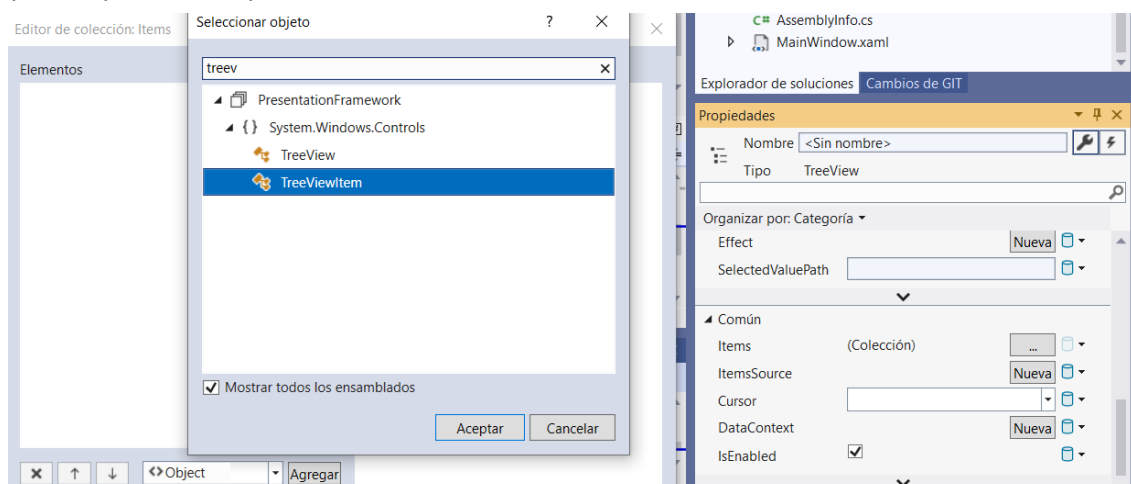
Id	Name	Birthday
1	John Doe	7/23/1971 12:00:00 AM
2	Jane Doe	1/17/1974 12:00:00 AM
3	Sammy Doe	9/2/1991 12:00:00 AM

5. TreeView

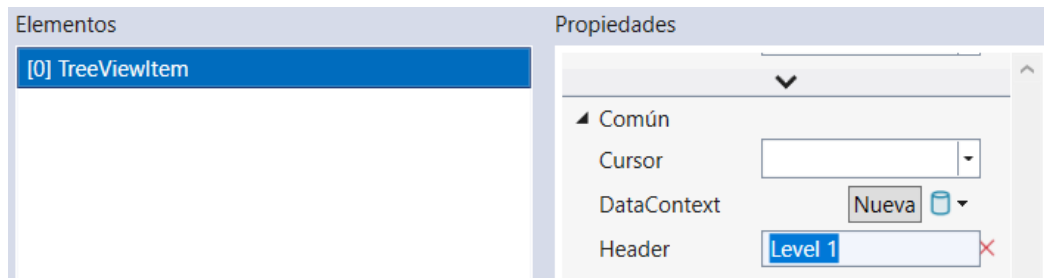
El control **TreeView** permite mostrar jerarquías, en donde cada dato es representado por un nodo en el árbol. Cada nodo puede tener hijos y cada hijo puede tener otro hijo y así sucesivamente como se puede observar en la aplicación de debajo.



Para crear un **TreeView**, arrastramos el control correspondiente. La forma más sencilla de añadir ítems sería desde la propiedad **Items** del **TreeView**. Aquí curiosamente no nos aparece el tipo **TreeViewItem** necesario para este control, aunque lo podemos buscar seleccionando **Object** tal y como puedes comprobar a continuación.

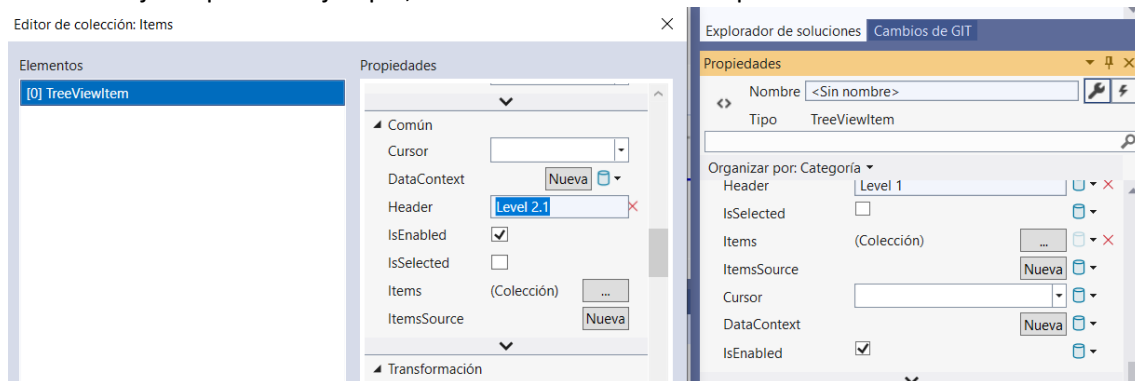


Asignamos el texto en el campo **Header**.



Además, desde el propio **TreeViewItem** podemos marcar otras propiedades como **IsExpanded** o **IsSelected** que más adelante veremos que resultan de utilidad.

Un **TreeView** es una estructura en forma de árbol y cada ítem contiene a otros. Por tanto, si queremos crear nuevos niveles nos tenemos que posicionar en el ítem a partir del cual vamos a añadir una jerarquía. Por ejemplo, en el elemento anterior es posible añadir más **TreeViewItem**.

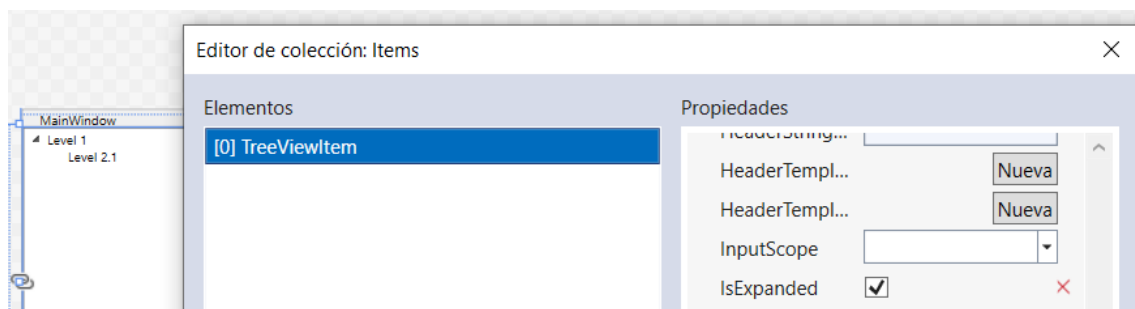


En la imagen de arriba, puedes comprobar que el nuevo ítem se está creando en la propiedad **Items** el primer **TreeViewItem**. El resultado será el siguiente.

```

└─ Level 1
    └─ Level 2.1
  
```

Y así sucesivamente por cada nivel que queramos añadir. Los ítems aparecen sin expandir por defecto. Como se explicaba previamente, podemos mostrar el contenido de un nivel con la propiedad **IsExpanded**. Por ejemplo, la asignamos en el ítem raíz "Level 1" y automáticamente mostrará el contenido en la previsualización.



Aunque añadir ítems es relativamente sencillo desde la interfaz gráfica, en este control puede que te resulte aún más fácil desde el propio XAML, ya que podemos tener una visión más clara de la jerarquía. A continuación se muestra un ejemplo con 3 niveles:

```
<TreeView>
  <TreeViewItem Header="Level 1" IsExpanded="True">
    <TreeViewItem Header="Level 2.1" />
    <TreeViewItem Header="Level 2.2" IsExpanded="True">
      <TreeViewItem Header="Level 3.1" />
      <TreeViewItem Header="Level 3.2" />
    </TreeViewItem>
    <TreeViewItem Header="Level 2.3" />
  </TreeViewItem>
</TreeView>
```

Cada etiqueta **TreeViewItem** contiene los ítems del nivel posterior y algunos aparecen expandidos por defecto. El resultado sería el siguiente:

```

└─ Level 1
    └─ Level 2.1
    └─ Level 2.2
        └─ Level 3.1
        └─ Level 3.2
    └─ Level 2.3
```