**MVVM – CALIBURN.MICRO – WPF**

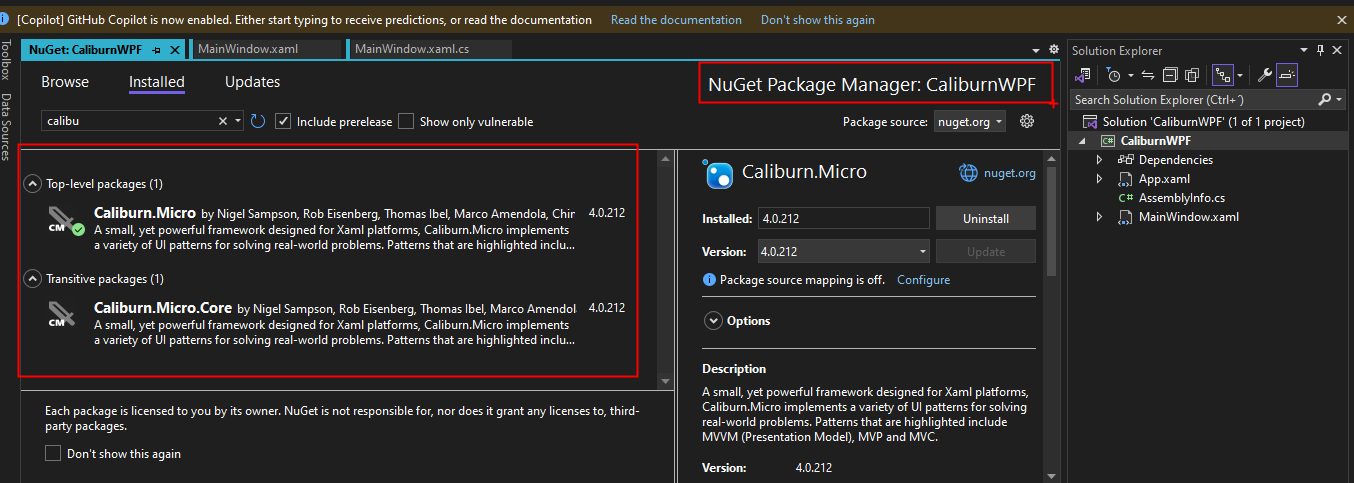
**Ojo es muy importante respetar la convención de los nombres de las carpetas de Models, Views y ViewModels**

**Enlace: ->** <https://www.youtube.com/watch?v=laPFq3Fhs8k&t=3276s>

**El patrón MVVM (Model-View-ViewModel)** es un patrón de diseño utilizado en el desarrollo de aplicaciones para separar la lógica de la interfaz de usuario de la lógica de negocio, facilitando así el mantenimiento y la escalabilidad del código. MVVM es especialmente popular en el desarrollo de aplicaciones WPF (Windows Presentation Foundation) y otras tecnologías basadas en XAML.

**Conceptos Clave de MVVM**

1. **Model**: Representa la lógica de negocio y los datos de la aplicación. No debe tener ninguna referencia a la interfaz de usuario.
2. **View**: Representa la interfaz de usuario. Contiene los controles y su disposición visual. En WPF, esto se define principalmente en archivos XAML.
3. **ViewModel**: Actúa como un intermediario entre el Model y la View. Contiene la lógica de presentación y maneja la comunicación entre el Model y la View. Implementa la interfaz INotifyPropertyChanged para notificar a la View sobre los cambios en los datos.

Instalamos la dependencia **CaliburnMicro**

Eliminamos el MainWindow.Xaml

Y ahora nos dirigimos a la clase **App.xml**  y pegamos el siguiente código y también eliminamos de ahí la referencia al **MainWindow.xml**:

<Application.Resources>

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary>

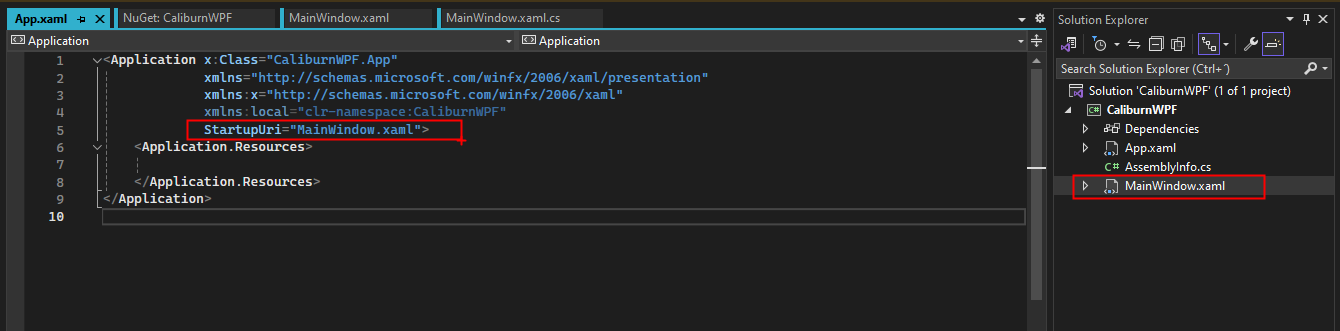
<local:Bootstrapper x:Key="bootstrapper" />

</ResourceDictionary>

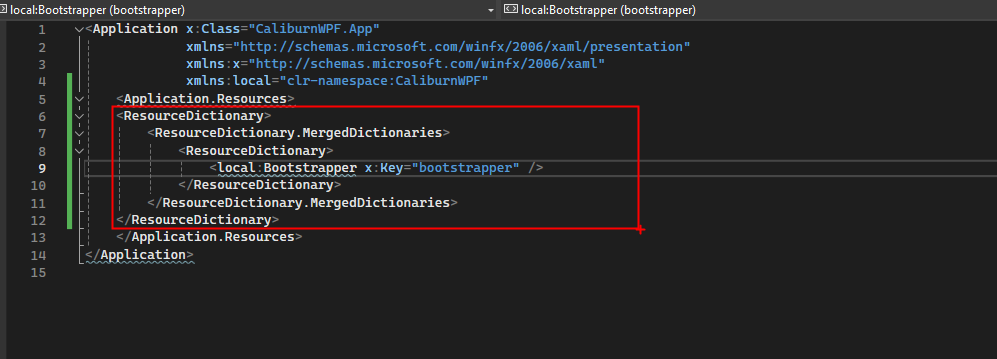
</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

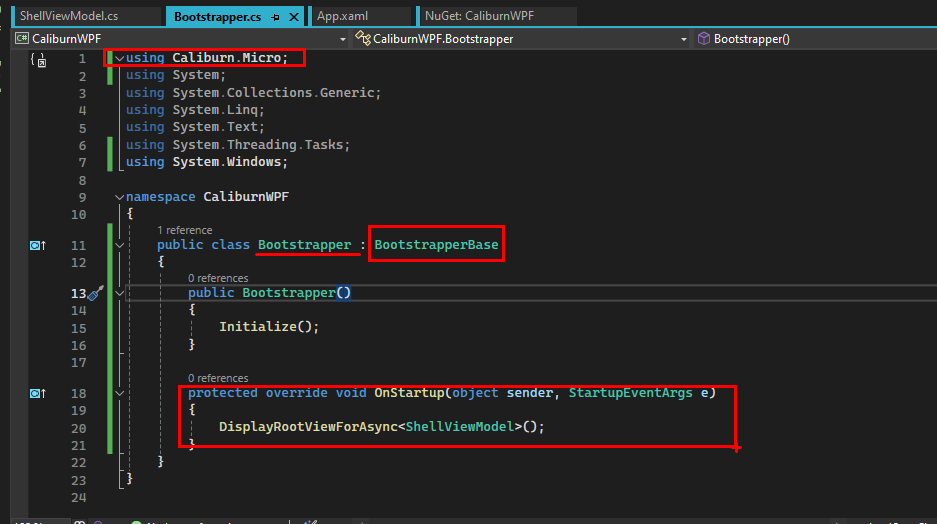
</Application.Resources>



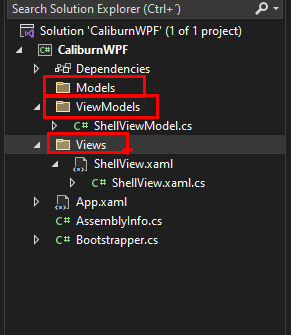
Agregamos el siguiente código:



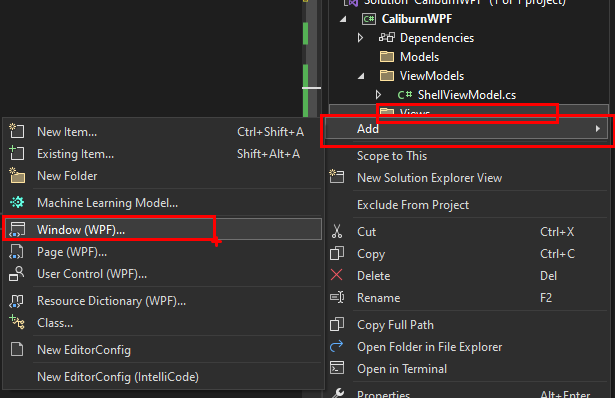
Ahora creamos nuestra clase **Bootstrapper**:



Creamos esta estructura de carpetas: es necesario respetar el nombre de las carpetas, ya que estos nombres de estas carpetas representan la estructura de nombres que utiliza la librería **Caliburn.Micro**



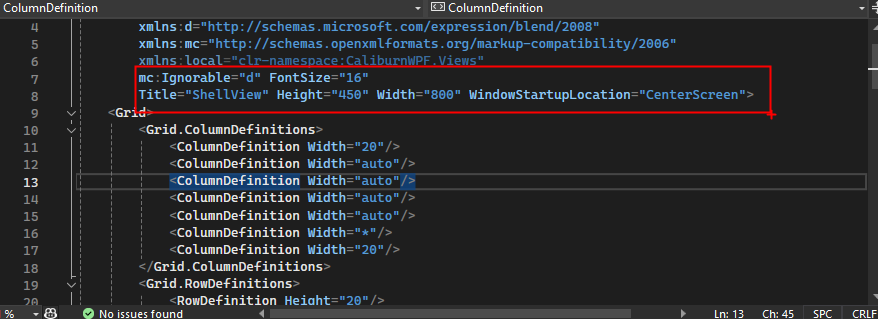
En Views, agregamos una nueva **ventana(Window WPF):** de nombre **ShellView**

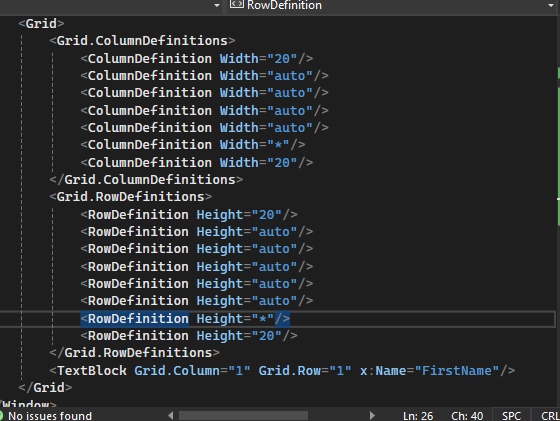


Y creamos una clase **ShellViewModel** en la carpeta **ViewModels**

Y ejecutamos el proyecto

Ahora creamos lo siguiente:





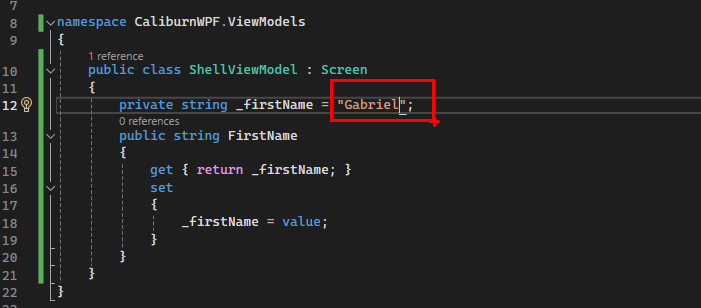
**SCREEN**

En Caliburn.Micro, la clase Screen es una implementación base del patrón MVVM para ViewModels que tienen una vida útil "activa" en la aplicación. Al heredar de Screen, tu ShellViewModel obtiene varias características y comportamientos que facilitan la creación de aplicaciones MVVM en WPF. Aquí hay algunas razones y beneficios de heredar de Screen:

**Beneficios de Heredar de Screen**

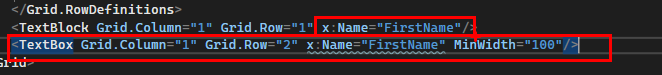
1. **Notificaciones de Cambio de Propiedad**: Screen implementa la interfaz INotifyPropertyChanged, lo que permite que las propiedades del ViewModel notifiquen a la vista cuando sus valores cambian. Esto es esencial para la vinculación de datos en WPF.
2. **Ciclo de Vida de la Pantalla**: Screen proporciona métodos y eventos para gestionar el ciclo de vida del ViewModel, como OnActivate, OnDeactivate, CanClose, etc. Esto es útil para realizar tareas específicas cuando el ViewModel se activa o desactiva.
3. **para Validación**: Screen también puede ayudar a gestionar la validación de datos en el ViewModel, lo cual es útil para asegurar que los datos ingresados por el usuario sean correctos.
4. **Gestión de Estados**: Screen facilita la gestión de estados como IsActive y IsInitialized, que son útiles para saber si un ViewModel está actualmente activo o ha sido inicializado.

Ahora actualizamos este campo: **FirstName**:



Ejecutamos el proyecto y vemos los cambios

Ahora creamos un textBox con esta propiedad igual de **FirstName** y borramos dicha propiedad de mi elemente **TextBlock**



Y ejecutamos

Luego vamos a ver el dataBinding:

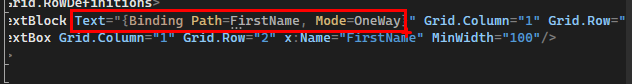
### Funcionamiento del Binding

El Binding en WPF permite que la propiedad Text del TextBlock esté enlazada a la propiedad FirstName del objeto de contexto de datos. Este objeto de contexto de datos es típicamente el ViewModel en el patrón MVVM. Aquí hay un ejemplo de cómo se vería esto en un escenario completo.

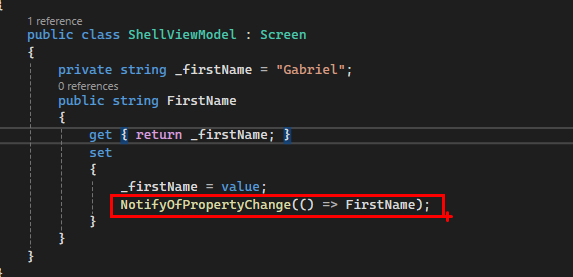
**Modos de Binding**

Además de OneWay, WPF soporta otros modos de binding:

* **OneWay**: La vista se actualiza cuando el ViewModel cambia, pero no al revés.
* **TwoWay**: La vista y el ViewModel se actualizan mutuamente cuando cualquiera de los dos cambia.
* **OneTime**: La vista se actualiza solo una vez cuando se carga por primera vez.
* **OneWayToSource**: Solo el ViewModel se actualiza cuando la vista cambia, no al revés.

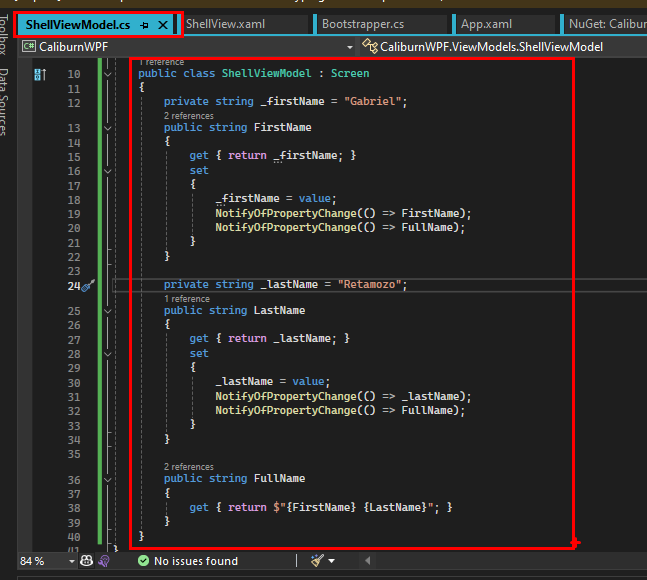
Ahora vamos a agregar esto en nuestro **TextBlock: **

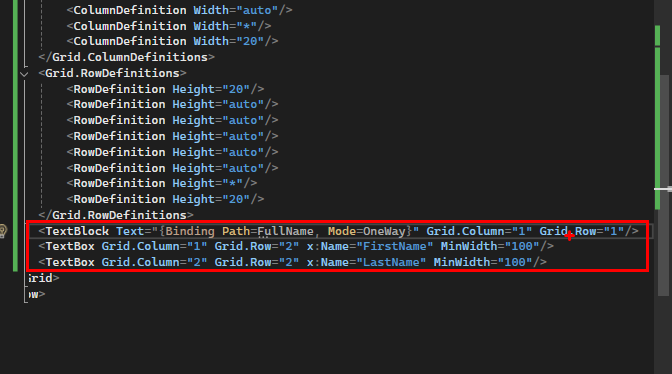
Ahora agregamos el **NotifyPropertyChange** en nuestro **ShellViewModel:**

****

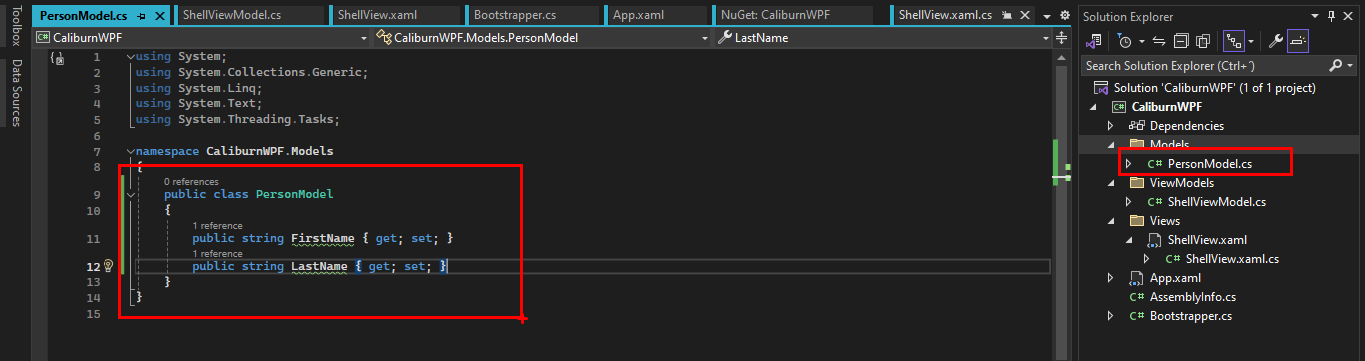
Y probamos, ejcutamos.

Luego aumentamos este código en nuestra clase **ShellViewModel**:





Ahora en nuestra carpeta **Models** agregamos esta clase **PersonModel:**

****

**BindableCollection**

es una clase proporcionada por el framework Caliburn.Micro, una librería popular para implementar el patrón MVVM en aplicaciones WPF y Silverlight. BindableCollection es una implementación de ObservableCollection, pero con algunas mejoras y características adicionales para facilitar el desarrollo de aplicaciones MVVM.

**Propósitos y Beneficios de BindableCollection**

1. **Notificación de Cambios**:
   * BindableCollection implementa INotifyCollectionChanged e INotifyPropertyChanged, lo que significa que cualquier cambio en la colección (añadir, eliminar, mover elementos) o en las propiedades de los elementos de la colección se notifica automáticamente a la vista.
   * Esto es crucial para mantener la interfaz de usuario sincronizada con los datos del ViewModel.
2. **Mejor Rendimiento**:
   * BindableCollection puede ofrecer mejoras de rendimiento sobre ObservableCollection en ciertos escenarios debido a optimizaciones internas específicas de Caliburn.Micro.
3. **Métodos Adicionales**:
   * BindableCollection añade algunos métodos útiles que no están presentes en ObservableCollection, facilitando tareas comunes de manipulación de colecciones.

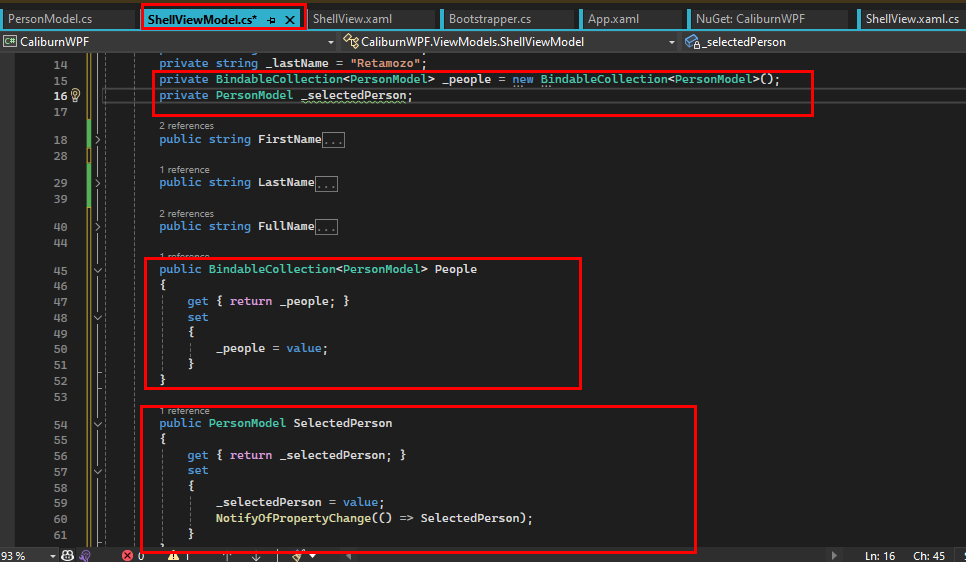
ObservableCollection

es una clase en el espacio de nombres System.Collections.ObjectModel en .NET que representa una colección de datos dinámica que proporciona notificaciones cuando los elementos se agregan, eliminan o cambian. Es ampliamente utilizada en aplicaciones WPF y Silverlight para implementar el patrón MVVM porque facilita la actualización automática de la interfaz de usuario cuando los datos cambian.

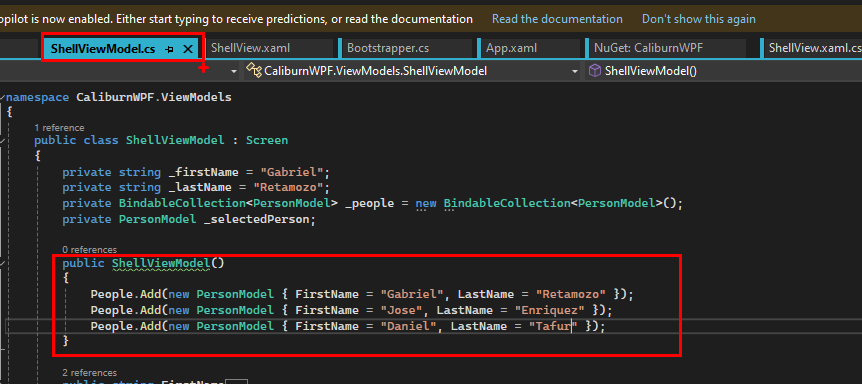
**Características Principales de ObservableCollection**

1. **Notificaciones de Cambios**:
   * Implementa la interfaz INotifyCollectionChanged, lo que significa que cualquier cambio en la colección (añadir, eliminar, mover elementos) notifica automáticamente a cualquier controlador de eventos que esté suscrito, como una vista en WPF.
   * También implementa INotifyPropertyChanged, que notifica cambios en las propiedades de los elementos de la colección.
2. **Actualización Automática de la UI**:
   * En aplicaciones WPF, cuando se enlaza una ObservableCollection a un control de interfaz de usuario como un ListBox o un DataGrid, la interfaz de usuario se actualiza automáticamente cuando la colección cambia.

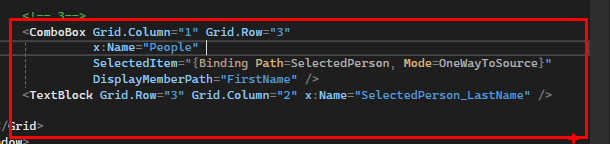
Ahora en mi clase de ShellViewModel aumentos lo siguiente:



Y por último agregamos un constructor para simular una base de datos:



Agregamos el siguiente código:



ComboBox:

 **x:Name="People"**:

* En Caliburn.Micro, el uso del nombre People para el ComboBox tiene un propósito especial. El framework asocia automáticamente el nombre del control con una propiedad en el ViewModel.
* En este caso, x:Name="People" hace que el ComboBox se vincule a la propiedad People del ViewModel. Esta propiedad es de tipo BindableCollection<PersonModel>.

 **ItemsSource**:

* Aunque no está explícitamente definido en el XAML, Caliburn.Micro automáticamente enlaza la propiedad ItemsSource del ComboBox con la propiedad People del ViewModel debido al nombre coincidente.
* Esto significa que los elementos del ComboBox se llenan con la colección de PersonModel en la propiedad People.

 **SelectedItem="{Binding Path=SelectedPerson, Mode=OneWayToSource}"**:

* Esta línea establece un binding bidireccional (por defecto) entre la propiedad SelectedItem del ComboBox y la propiedad SelectedPerson del ViewModel.
* SelectedItem se vincula a SelectedPerson, y cualquier cambio en la selección del ComboBox actualizará SelectedPerson en el ViewModel.
* Mode=OneWayToSource significa que solo los cambios del ComboBox se enviarán al ViewModel, pero no al revés.

 **DisplayMemberPath="FirstName"**:

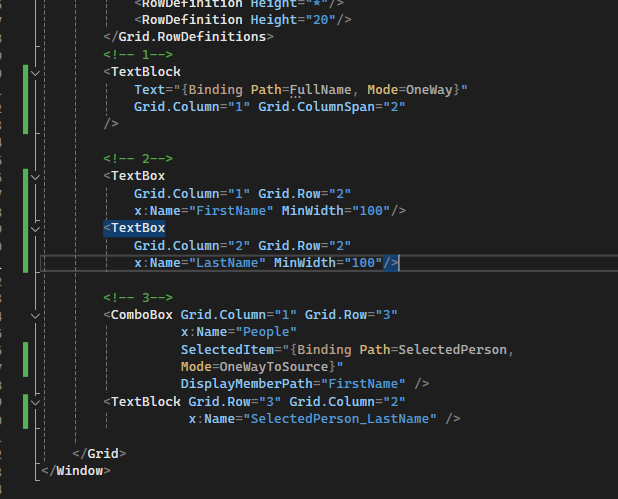
* Define qué propiedad del objeto PersonModel se mostrará en el ComboBox.
* En este caso, el ComboBox mostrará la propiedad FirstName de cada PersonModel en la colección People.

TextBox:

**x:Name="SelectedPerson\_LastName"**:

* El nombre del TextBlock SelectedPerson\_LastName sigue la convención de <NombrePropiedad>\_<NombreSubpropiedad>.
* Caliburn.Micro interpreta esto y enlaza automáticamente el TextBlock a la propiedad LastName de la propiedad SelectedPerson en el ShellViewModel.

Ahora completamos los respectivos **bindings:**

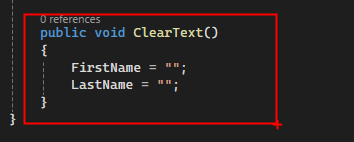


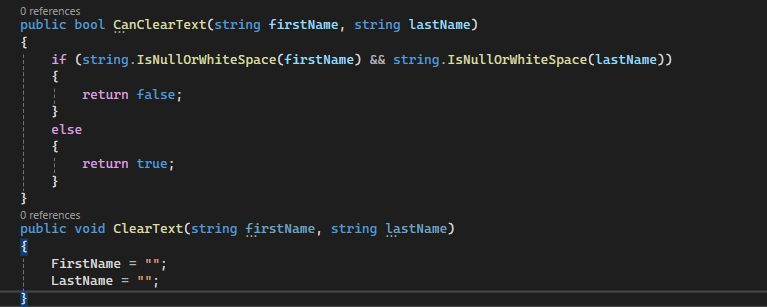
En ambos casos, FullName es la propiedad de origen de datos a la que se está enlazando el TextBlock. La propiedad Path en la expresión de enlace es opcional cuando estás enlazando directamente a una propiedad. El comportamiento y el resultado serán los mismos en ambos ejemplos.

El uso de Path se vuelve más relevante cuando necesitas enlazar a propiedades anidadas o utilizar una sintaxis más compleja,

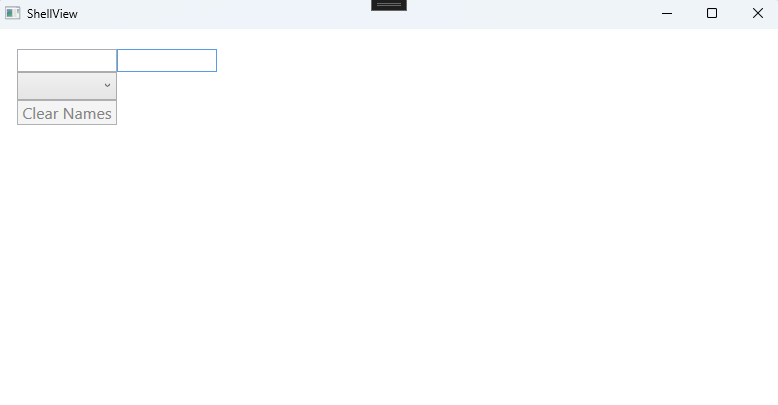
**En el contexto de Caliburn.Micro, el botón <Button x:Name="ClearText"> se relaciona automáticamente con el método público ClearText en tu ShellViewModel. Este enlace se realiza debido a la convención de nombrado en Caliburn.Micro, donde los controles en la vista (XAML) con el prefijo x:Name intentan enlazarse a los métodos con el mismo nombre en el ViewModel.**

Y ahora agregamos esto:





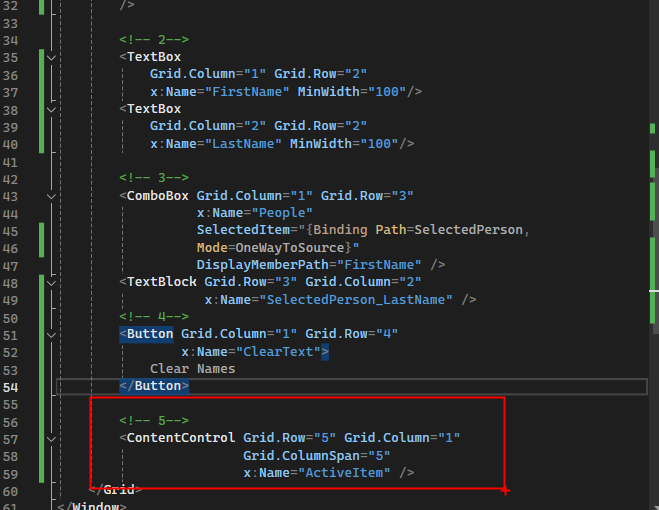
Y comprobamos así, que todo este vacio:



**CREAMOS UN NUEVO PROYECTO O PODEMOS COIPAR ESTE PROYECTO DE NUEVO**

Ya que usaremos en vez de **Screen** usaremos **Conductor<object>**

**Ahora agregamos un ContentControl:**

****

**ContentControl**

En WPF (Windows Presentation Foundation), un ContentControl es un control que puede contener un solo elemento de contenido de cualquier tipo. Esto puede ser un control simple como un botón, una imagen o un texto, o un elemento más complejo como un panel de otros controles.

El ContentControl puede contener cualquier tipo de contenido. En el contexto de Caliburn.Micro, este control se utiliza a menudo para mostrar la vista activa en una aplicación que sigue el patrón MVVM. Caliburn.Micro tiene una característica llamada Conductor que se utiliza para gestionar las vistas y ViewModels activos. Cuando cambias la propiedad ActiveItem en tu ViewModel, Caliburn.Micro automáticamente actualiza el contenido del ContentControl para mostrar la vista correspondiente al ViewModel activo.

### Resumen

El ContentControl en tu ejemplo se utiliza para mostrar dinámicamente el contenido que cambia en tu aplicación. Cuando la propiedad ActiveItem cambia en el ViewModel, Caliburn.Micro actualiza el contenido del ContentControl para mostrar la vista correspondiente al ViewModel activo. Esto es muy útil para aplicaciones que necesitan cambiar vistas dinámicamente sin necesidad de escribir mucho código adicional para manejar el cambio de vistas.

Ahora ya que vamos a implementar un **ControlContent** es necesario cambiar **“ : Screen”** por **Conductor<object>**

Conductor<object> es una clase de Caliburn.Micro. Es parte del framework y está diseñada para gestionar otros ViewModels (o "items"). Esta clase facilita la gestión de la navegación y la composición de múltiples pantallas dentro de una aplicación. Aquí hay una explicación más detallada sobre qué es y cómo se usa Conductor<object> en Caliburn.Micro.

**Conductor en Caliburn.Micro**

Un Conductor en Caliburn.Micro es un tipo de Screen que puede "conducir" o gestionar uno o más "items". Estos items pueden ser otros ViewModels. El Conductor se encarga de activar, desactivar y gestionar el ciclo de vida de estos items.

**Tipos de Conductores**

Caliburn.Micro ofrece varios tipos de Conductors:

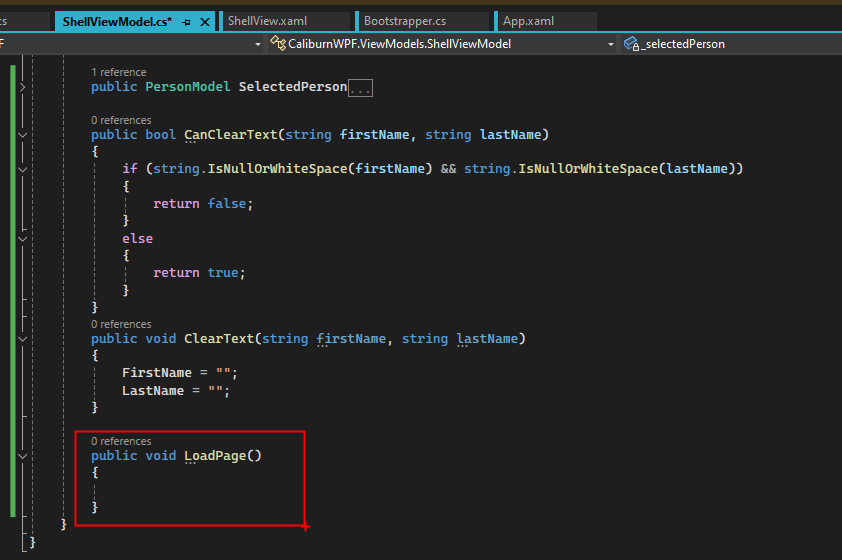
1. **Conductor<T>**: Este es el conductor base que puede gestionar items de tipo T.
2. **Conductor<IScreen>**: Gestiona items que implementan la interfaz IScreen.
3. **Conductor<object>**: Gestiona items de tipo object, lo que le permite manejar cualquier tipo de ViewModel.

**Ejemplo: Uso de Conductor<object>**

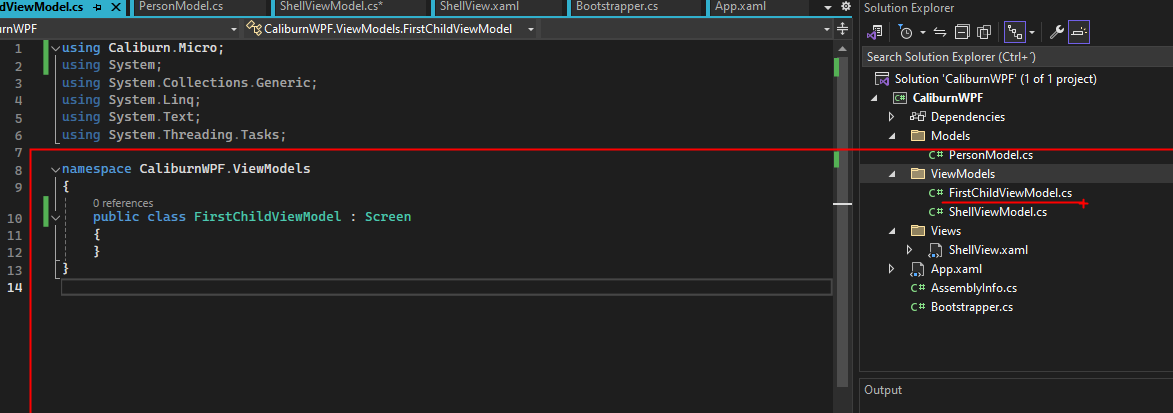
Supongamos que tienes una aplicación donde deseas navegar entre diferentes pantallas o secciones. Usar Conductor<object> te permitirá gestionar estos cambios de forma eficiente.

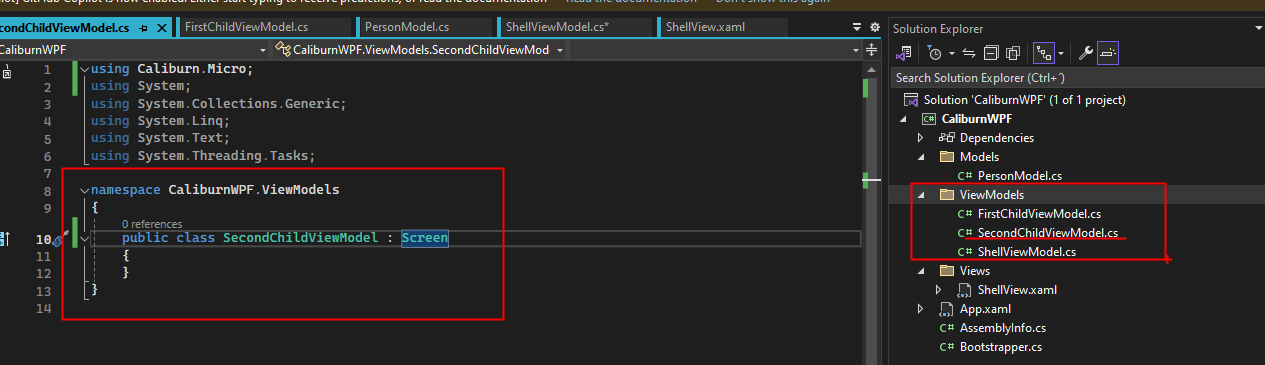
Heredar de Conductor<object> en Caliburn.Micro permite a tu ViewModel gestionar y controlar múltiples ViewModels secundarios. Esto es útil para aplicaciones con navegación compleja o múltiples pantallas, ya que Conductor maneja automáticamente el ciclo de vida de estos ViewModels, simplificando así la lógica de navegación y composición de la UI.

Y ahora agregamos este método **LoadPage()**

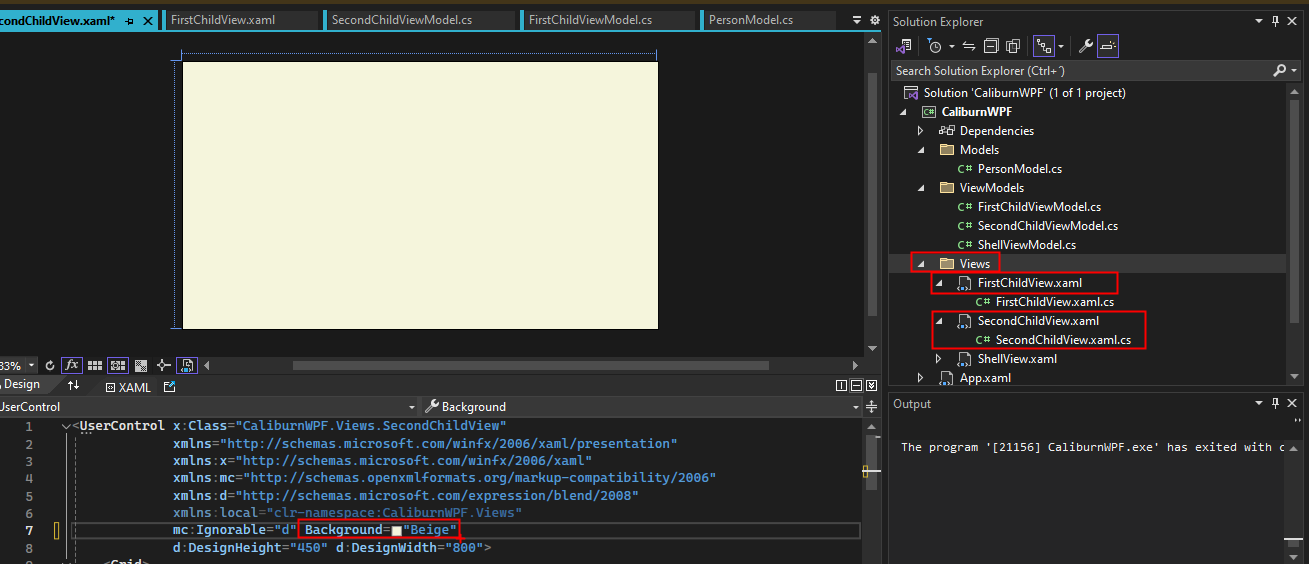
****

Ahora seguidamente vamos a agregar dos **ViewModels**(clases) hijas en nuestra carpeta **ViewModel**





Y ahora agregamos sus respectivos **UserControls**:



### Nueva Ventana (Window)

#### Características:

1. **Ventana Separada**: Cada Window es una ventana independiente que puede tener su propio borde, barra de título y botones de control (minimizar, maximizar, cerrar).
2. **Contexto de Aplicación Independiente**: Una nueva ventana puede tener su propio contexto de datos y lógica independiente del resto de la aplicación principal.
3. **Interacción del Usuario**: Cada ventana puede ser movida, redimensionada y gestionada de forma independiente.
4. **Gestión de Ciclo de Vida**: Abrir y cerrar ventanas implica gestionar su ciclo de vida, lo que puede incluir la creación de nuevas instancias, el manejo de eventos de cierre, etc.

#### Uso Apropiado:

* Cuando necesitas una ventana completamente separada de la ventana principal, por ejemplo, para configuraciones, diálogos, formularios de entrada de datos independientes, etc.
* Cuando quieres que el usuario pueda trabajar con varias ventanas simultáneamente.

### Control (UserControl)

#### Características:

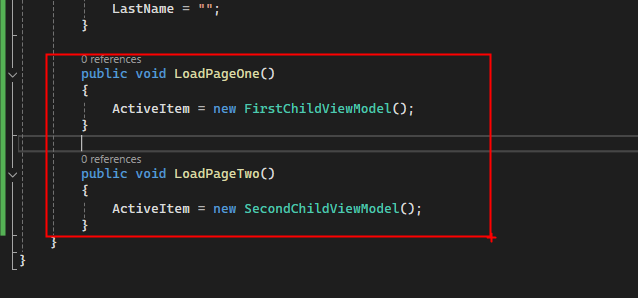
1. **Parte de una Ventana Existente**: Un UserControl es un componente reutilizable que se integra dentro de una ventana existente (o dentro de otros controles). No tiene su propio borde o barra de título.
2. **Reutilización y Composición**: Los UserControl son ideales para crear componentes reutilizables que pueden ser utilizados en múltiples lugares dentro de una aplicación.
3. **Contexto de Datos Común**: Generalmente, un UserControl comparte el contexto de datos y la lógica de la ventana que lo contiene.
4. **Gestión Simplificada**: No se necesita gestionar ventanas independientes, lo que simplifica la lógica de la interfaz de usuario y la navegación.

#### Uso Apropiado:

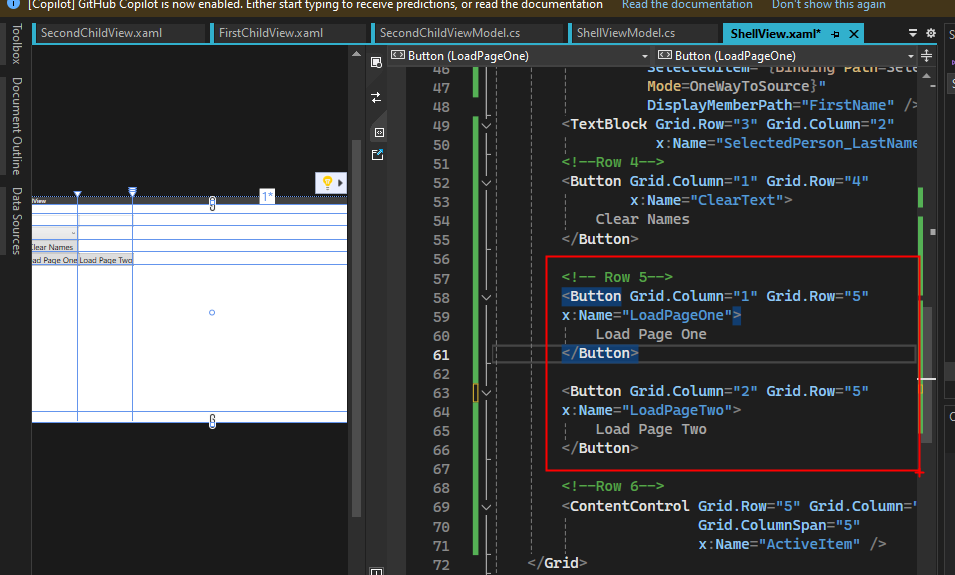
* Cuando necesitas crear componentes reutilizables y modulares dentro de una ventana existente.
* Para organizar y estructurar la interfaz de usuario dentro de una única ventana principal.
* Cuando quieres mantener el contexto de datos y la lógica compartidos entre múltiples componentes dentro de una ventana.

En resumen, la elección entre usar una nueva ventana o un control depende de la necesidad de independencia y reutilización en tu aplicación. Las nuevas ventanas son ideales para tareas independientes y contextos de usuario separados, mientras que los controles son mejores para crear interfaces modulares y reutilizables dentro de una ventana existente.

Ahora implementamos estos dos métodos, que estarán asociados a los dos eventos d mis dos botones:



Y agregamos estos botones en mi **ShellView:**

****

Y ejecutamos la aplicación

Debería estar asi:

