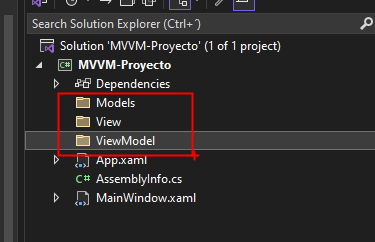
**Proyecto con MVVM en WPF**

**Enlace: ->** <https://www.youtube.com/watch?v=1_HsXz7mxhA>

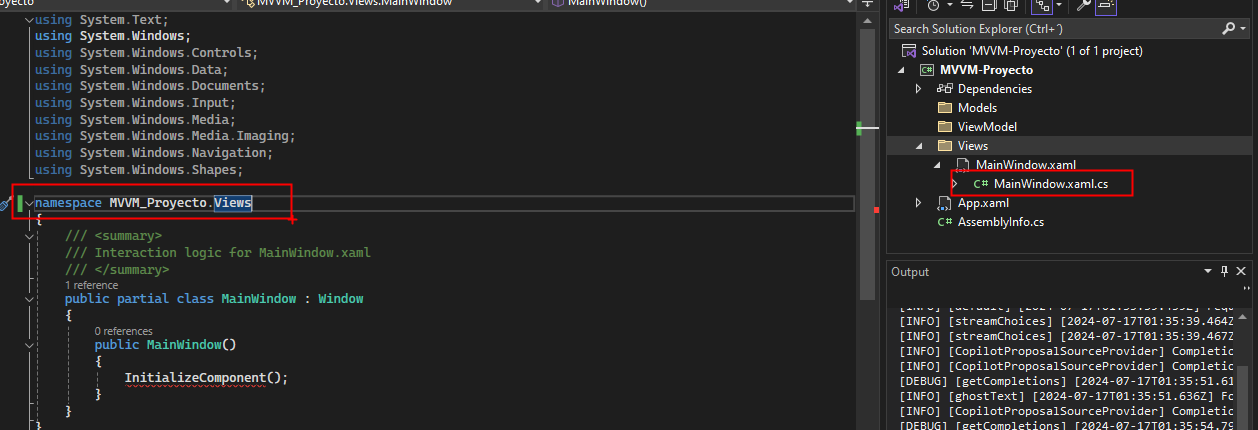
Creamos un Nuevo proyecto en WPF

Creamos las carpetas: **View**, **Model** y **ViewModel**

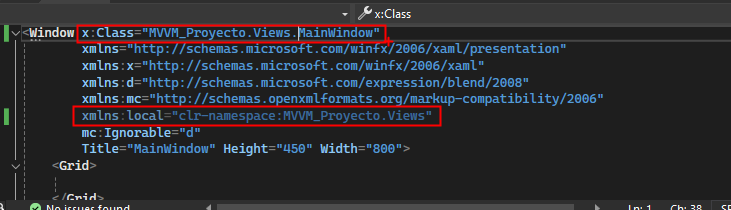


Y pasamos nuestro archivo **MainWindow.xaml** que representa mi vista a la carpeta **View**

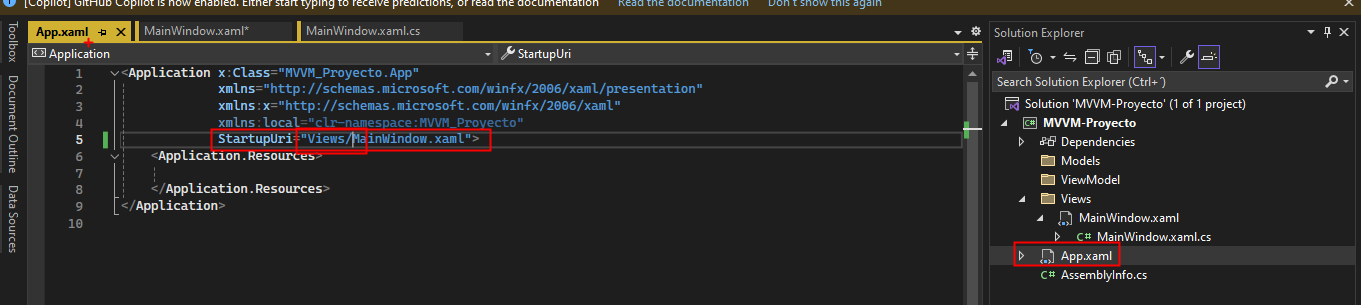
Y reajustamos el **nameSpace** de nuestro archivo **MainWindow.xaml.cs**



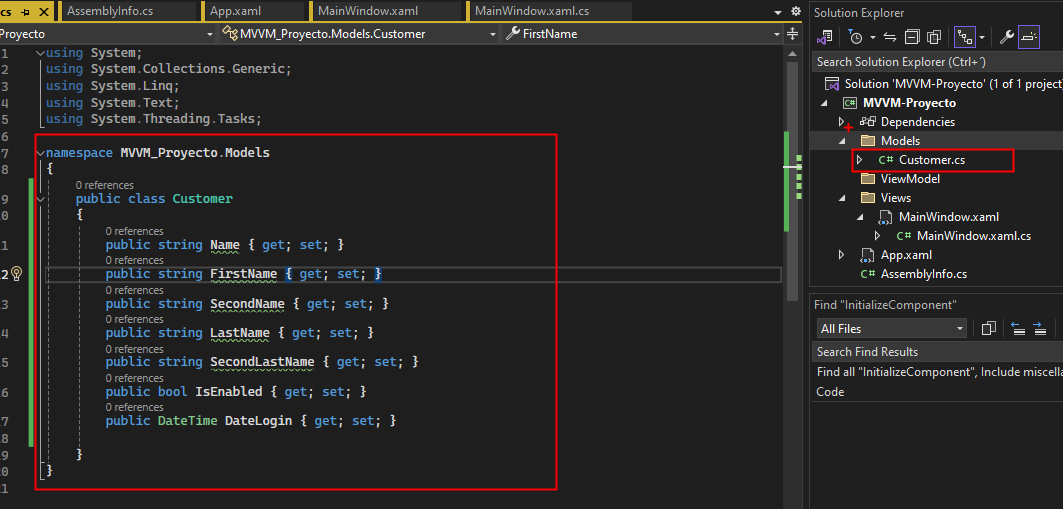
Luego tengo que modificar mi diseñador **Xaml:**



Y en el archivo **App.xaml** tenemos que modificar:



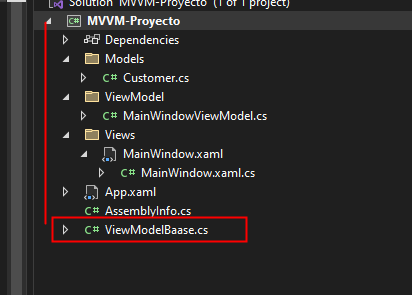
Y ahora procedemos a crear nuestra clase **Customer** dentro de la carpeta **Model:**

****

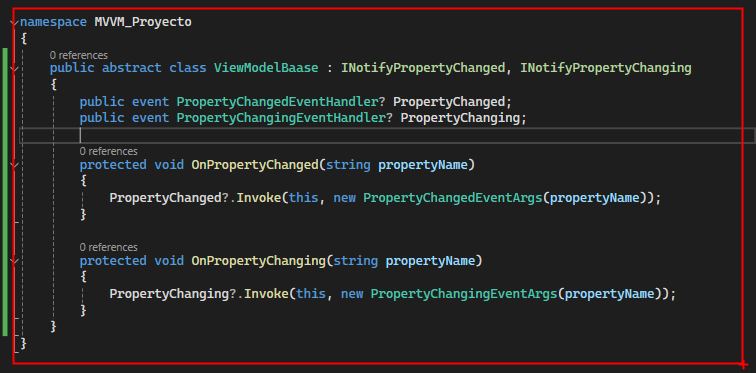
Ahora creamos otra clase dentro de la carpeta **ViewModel** la cual tendrá por nombre **MainWindowViewModel** y tendrá la función de tener toda la lógica de la aplicación.

Esta clase **MainWindowViewModel** heredara de una clase **ViewModelBase** que también crearemos ahora,

Entonces creamos una clase **abstracta** de nombre **ViewModelBase** por fuera de las carpetas creadas a esta altura:



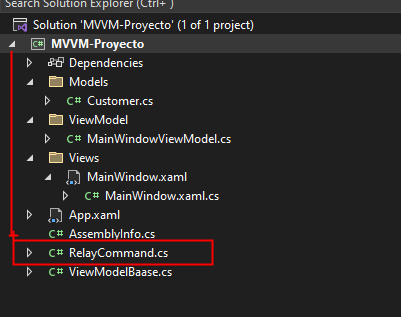
Y hacemos la implementación:

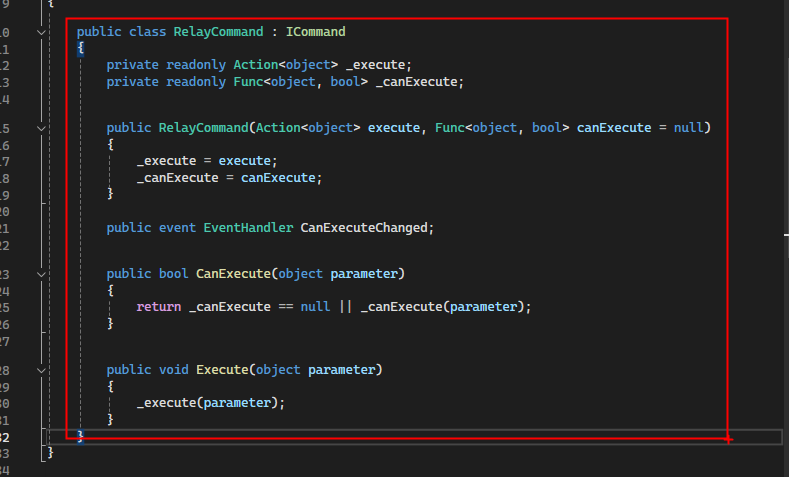


Lo cual es necesario para que puede usarse en mi clase **MainWindowViewModel**

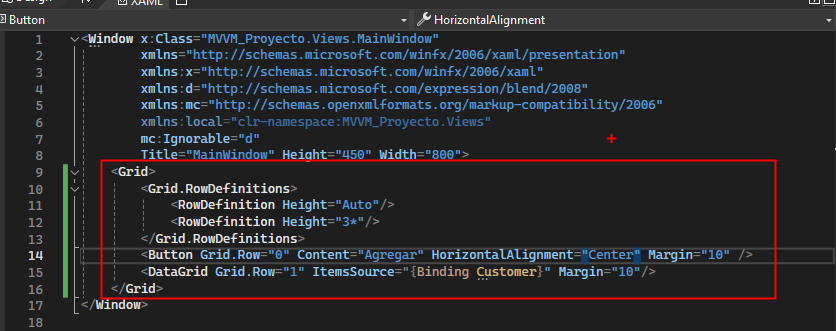
Esta clase **ViewModelBase** no es más que declarar una clase base que va a heredar de dos interfaces muy importantes a la hora de cambiar una propiedad dentro de una vista que es **INotifyPropertyChanged** y **INotifyPropertyChanging** ya que notificamos a la vista que ocurrió un cambio.

Seguidamente tenemos que crear nuestra clase **RelayCommand** que será necesario de igual forma para mi **MainWindowViewModel**:



Y ponemos el siguiente código: 

Ahora implementamos la vista:



El control DataGrid en WPF es un control muy utilizado para mostrar y manipular datos tabulares. Aquí hay un resumen de lo que hace y cómo se usa:

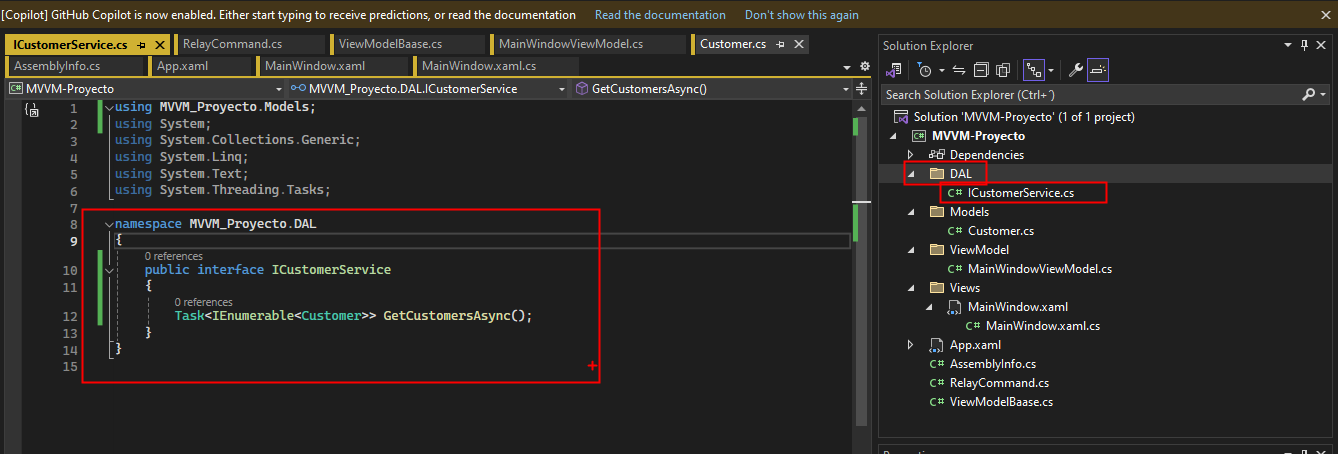
**Funcionalidad del DataGrid**

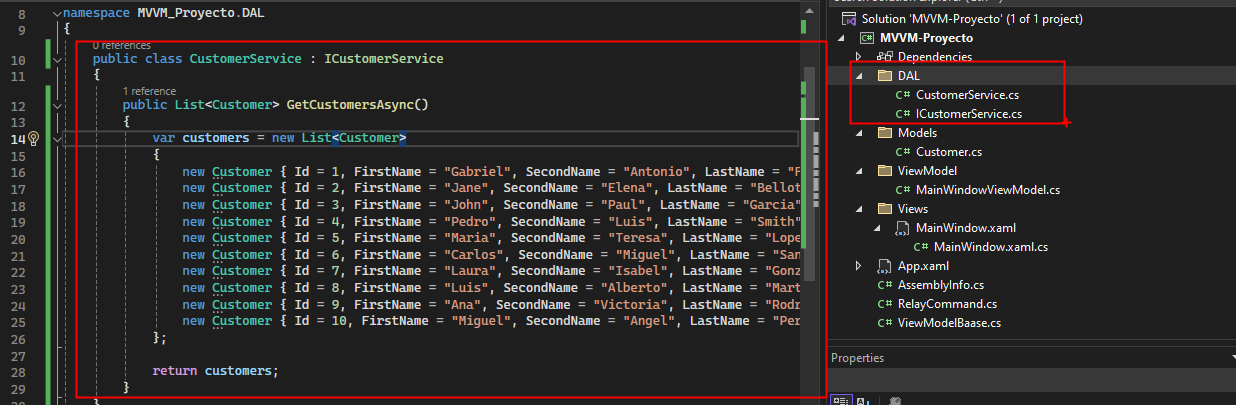
El DataGrid permite:

1. **Mostrar Datos Tabulares**: Muestra datos en un formato de tabla con filas y columnas.
2. **Edición de Datos**: Permite la edición de datos directamente en la interfaz de usuario.
3. **Ordenación**: Proporciona soporte para ordenar datos en base a las columnas.
4. **Filtrado y Agrupación**: Aunque no está soportado de manera nativa, se puede extender para soportar filtrado y agrupación.
5. **Selección de Filas**: Permite la selección de una o más filas.
6. **Personalización**: Se puede personalizar mediante estilos y plantillas para cambiar su apariencia y comportamiento.

Seguidamente vamos a crear una clase que simulara nuestra base de datos, que se llamara **ICustomerService**:

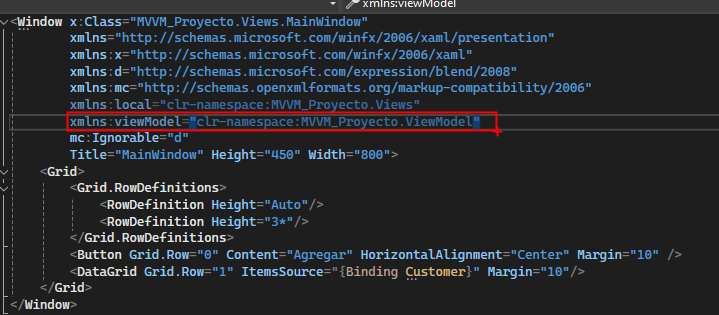
Para ello primeramente crearemos una carpeta de nombre **DAL** la cual contendrá la interfaz y su implementación, esta carpeta estará por fuera de las demás carpetas creadas:





Ahora si pasamos a desarrollar la implementación como tal de **MVVM:**

Primeramente, tenemos que traer nuestra clase **MainWindowViewMode** que se encuentra dentro de la carpeta **ViewModel:**

****

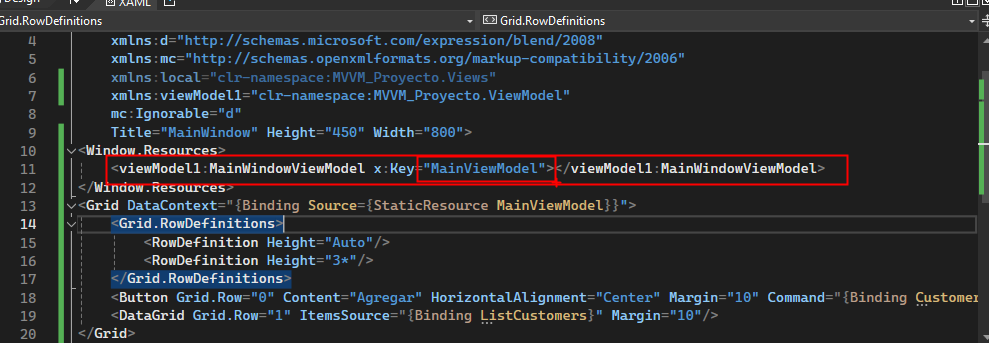
El código xmlns:viewModel="clr-namespace:MVVM\_Proyecto.ViewModel" en un archivo XAML hace lo siguiente:

1. **Define un espacio de nombres XML (XML Namespace):** Esta línea define un prefijo de espacio de nombres XML (viewModel) que se asocia con un espacio de nombres de CLR (Common Language Runtime) en el código C#.
2. **Asociación con un espacio de nombres CLR:** El valor "clr-namespace:MVVM\_Proyecto.ViewModel" especifica el espacio de nombres CLR dentro del cual se encuentran las clases que deseas usar en tu archivo XAML. En este caso, el espacio de nombres es MVVM\_Proyecto.ViewModel.

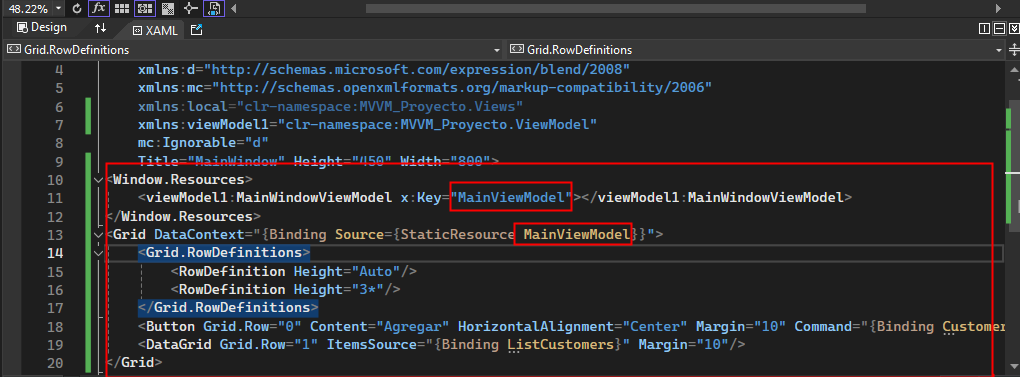
Supongamos que tienes una clase MainViewModel en tu espacio de nombres MVVM\_Proyecto.ViewModel y quieres usarla en tu XAML. Primero se define el espacio de nombres XML.

Ahora seguidamente para usar nuestra clase **MainWindowViewModel** tenemos que hacer uso de **Window.Resource:**

Y con nuestro **nameSpace** creado de nombre **viewModel1** llamamos a la clase que vamos a usar que es **MainWindowViewModel**, y posteriormente le indicamos la llave con la cual nos vamos a referir a ella, en este caso**: MainViewModel:**

****

Ahora en nuestro **Grid** usaremos la propiedad **DataContext** para que con el **binding** se pueda enlazar con dicha clave **MainViewModel:**

****

Con esto ya hicimos el enlace de la vista **MainWindow.xaml** a **ViewModel** y el **ViewModel** va a llamar cosas del **Model**

En el código <Grid DataContext="{Binding Source={StaticResource MainViewModel}}">, el atributo DataContext establece el contexto de datos para el Grid y sus elementos secundarios. Aquí se utilizan vinculaciones de datos (data binding) para conectar la interfaz de usuario con el ViewModel en el patrón MVVM.

Ahora procederemos a disparar un command en el button:

### Propósito del Command

El comando se utiliza para definir la lógica que debe ejecutarse cuando el usuario hace clic en el botón. En el patrón MVVM (Model-View-ViewModel), los comandos son una manera de desacoplar la lógica de la interfaz de usuario del código de la interfaz de usuario.

 Command="{Binding CustomerCommand}"

* Esto significa que el botón está vinculado a una propiedad CustomerCommand en el ViewModel. Cuando se hace clic en el botón, se ejecutará el comando al que está vinculado.

 **Implementación del Command en el ViewModel:**

* En el ViewModel, CustomerCommand debe ser una propiedad de tipo ICommand.

el Command en WPF actúa de manera similar a un evento, como el evento Click, pero tiene algunas ventajas y diferencias clave que lo hacen más adecuado para el patrón MVVM (Model-View-ViewModel).

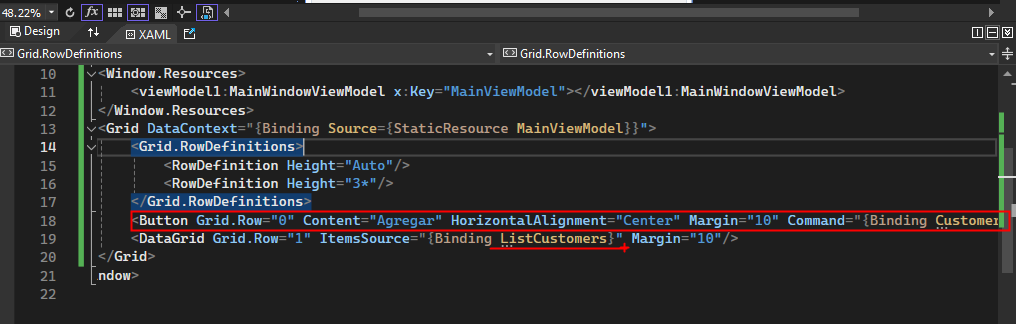
**Diferencias y Ventajas de los Commands sobre los Eventos**

1. **Desacoplamiento:**
   * **Eventos:** El controlador del evento Click se define en el code-behind (archivo .xaml.cs), lo que acopla la lógica de la interfaz de usuario con el código de presentación.
   * **Commands:** La lógica del comando se define en el ViewModel, lo que permite una separación más clara entre la interfaz de usuario (View) y la lógica de negocios/presentación (ViewModel).
2. **Reutilización:**
   * **Eventos:** Los controladores de eventos están típicamente asociados a un solo control.
   * **Commands:** Los comandos pueden ser reutilizados en múltiples controles. Por ejemplo, el mismo comando puede ser asociado a múltiples botones.

**Resumen**

* **Eventos:** Son más directos y fáciles de implementar en pequeñas aplicaciones, pero pueden llevar a un fuerte acoplamiento entre la lógica de negocios y la interfaz de usuario.
* **Commands:** Son más adecuados para aplicaciones que siguen el patrón MVVM, promoviendo una separación clara de responsabilidades, reutilización de lógica y mejor capacidad de pruebas.

El uso de Command es una práctica recomendada en WPF cuando se sigue el patrón MVVM, proporcionando una manera más estructurada y mantenible de manejar la lógica de interacción de la interfaz de usuario.



Posteriormente nos dirigimos a nuestra clase que representa nuestro **ViewModel** con el nombre de **MainWindowViewModel** y marcamos dos regiones la región de **properties** y la región de **comands**, seguidamente más abajo pondremos el constructor y los métodos necesarios, en ese orden:

public class MainWindowViewModel : ViewModelBaase

{

#region Properties

private List<Customer> \_listCustomers;

public List<Customer> ListCustomers

{

get { return \_listCustomers; }

set

{

if(\_listCustomers != value)

{

\_listCustomers = value;

OnPropertyChanged("ListCustomers");

}

}

}

#endregion

#region Commands

private ICommand \_customerCommand;

public ICommand CustomerCommand

{

get

{

if (\_customerCommand == null)

{

\_customerCommand = new RelayCommand(param => CustomerCommandExecute(), null);

}

return \_customerCommand;

}

}

#endregion

public MainWindowViewModel()

{

}

private void CustomerCommandExecute()

{

var customerService = new CustomerService();

var result = customerService.GetCustomers();

ListCustomers = new List<Customer>(result);

}

}

Y probamos nuestra aplicacion