



Jorge Quintana González

Introducción










Vamos a hacer un proyecto con una estructura modelo vista controlador

Objetivo

Aprender a trabajar en un proyecto con una estructura modelo vista controlador

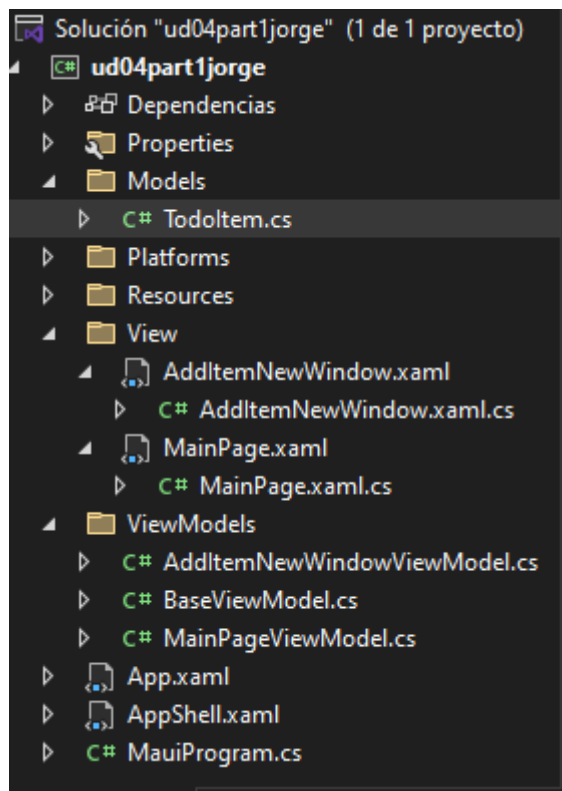
Material

He realizado este proyecto en una máquina virtual de windows con la siguiente características:

	General
Nombre:	Windows_VS2022
Sistema operativo:	Windows 10 (64-bit)
	Sistema
Memoria base:	9661 MB
Procesadores:	2
Orden de arranque:	Disquete, Óptica, Disco duro
EFI:	Habilitado
Aceleración:	Paginación anidada, PAE/NX, Paravirtualización Hyper-V
	Pantalla
Memoria de vídeo:	128 MB
Controlador gráfico:	VBoxSVGA
Servidor de escritorio remoto:	Inhabilitado
Grabación:	Inhabilitado
	Almacenamiento
Controlador:	SATA
Puerto SATA 0:	Windows_VS2022-disk001.vdi (Normal, 80,00 GB)
Puerto SATA 1:	[Unidad óptica] Vacío
	Audio
Controlador de anfitrión:	Predeterminado
Controlador:	Audio Intel HD
	Red
Adaptador 1:	Intel PRO/1000 MT Desktop (NAT)
	USB
Controlador USB:	xHCI
Filtros de dispositivos:	0 (0 activo)
	Carpetas compartidas
	Ninguno
	Descripción
	oscgaresc - 1234

Actividad

Cree la estructura del proyecto, siguiendo la arquitectura modelo vista controlador



Creo la clase TodoItem

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace ud04part1jorge.Models
8  {
9      10 referencias
10     public class TodoItem
11     {
12         3 referencias
13         public string Title { get; set; }
14         3 referencias
15         public bool IsCompleted { get; set; }
16     }
17 }

```

Este código describe una página donde el usuario puede añadir una nueva tarea a su lista de tareas. La página utiliza un `ContentPage`, y dentro de ella se organiza el contenido con un `VerticalStackLayout` para disponer los elementos de forma vertical. Primero, se establece el `BindingContext` de la página a un objeto del tipo `AddItemNewWindowViewModel`, lo que permite que la vista se vincule a las propiedades del `ViewModel` correspondiente.

Dentro del layout, se muestra un título en una etiqueta con el texto "Nueva tarea", centrado en la página. A continuación, se incluye un campo de entrada (`Entry`) donde el usuario puede escribir el nombre de la tarea, con un texto de placeholder que dice "Nombre de la tarea", y el valor del texto está vinculado a la propiedad `NewTaskTitle` en el `ViewModel`.

Después, se incluye un `CheckBox` que permite al usuario marcar si la tarea está completada o no. El estado del `CheckBox` se vincula a la propiedad `NewTaskIsCompleted` en el `ViewModel`, lo que significa que si el `CheckBox` está marcado, la propiedad será `true`, y si no lo está, será `false`.

Por último, se incluyen dos botones dentro de un `HorizontalStackLayout` para que estén alineados horizontalmente. El primer botón, con el texto "Añadir", está vinculado al comando `AddTaskCommand` del `ViewModel`, lo que ejecutará la lógica para añadir la tarea a la lista cuando se presione. El segundo botón, con el texto "Cancelar", está vinculado al comando `CancelCommand`, el cual cancelará la operación.

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:vm="clr-namespace:ud04part1jorge.ViewModels"
  x:Class="ud04part1jorge.View.AddItemNewWindow">
  <ContentPage.BindingContext>
    <vm:AddItemNewWindowViewModel />
  </ContentPage.BindingContext>

  <VerticalStackLayout Padding="20" Spacing="10">
    <Label Text="Nueva tarea"
      FontSize="24"
      HorizontalOptions="Center" />

    <!-- Entrada para el título de la tarea -->
    <Entry Placeholder="Nombre de la tarea"
      Text="{Binding NewTaskTitle}" />

    <!-- CheckBox para marcar si está completada -->
    <CheckBox IsChecked="{Binding NewTaskIsCompleted}" />
    <Label Text="¿Está completada?" />

    <!-- Botones para añadir o cancelar -->
    <HorizontalStackLayout Spacing="10">
      <Button Text="Añadir"
        Command="{Binding AddTaskCommand}" />
      <Button Text="Cancelar"
        Command="{Binding CancelCommand}" />
    </HorizontalStackLayout>
  </VerticalStackLayout>
</ContentPage>

```

Este código describe una página que muestra una lista de tareas y permite al usuario agregar nuevas tareas y eliminarlas. Utilizo un `ContentPage` para crear la interfaz de usuario, y en la parte superior de la página, se establece el `BindingContext` a un objeto del tipo `MainPageViewModel`, lo que permite que la vista se conecte con la lógica del `ViewModel`. La página tiene un `VerticalStackLayout`, que organiza los elementos verticalmente..

Al principio de la página, incluí una etiqueta con el texto "Lista de tareas", que está centrada. Debajo de esta etiqueta, agregué un botón que permite al usuario añadir una nueva tarea. El botón está vinculado al comando `AddItemCommandNewWindow` en el `ViewModel`, lo que ejecuta la lógica para navegar a una nueva página donde el usuario puede crear una nueva tarea.

Luego, tengo una `CollectionView`, que es el control que muestra las tareas. Cada tarea se muestra usando un `DataTemplate` que incluye un `StackLayout` horizontal con un `CheckBox`, una etiqueta que muestra el título de la tarea y un botón para eliminar la tarea. El `CheckBox` está vinculado a la propiedad `IsCompleted` de la tarea, lo que permite marcar o desmarcar la tarea como completada. Si la tarea está completada, el texto del título de la tarea se muestra con un efecto de tachado

gracias a un DataTrigger que cambia la propiedad TextDecorations de la etiqueta a Strikethrough.

Finalmente, hay un botón de eliminar asociado con cada tarea. Este botón está vinculado al comando DeleteItemCommand en el ViewModel, y permite eliminar la tarea cuando el usuario lo presiona.

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:vm="clr-namespace:ud04part1jorge.ViewModels"
x:Class="ud04part1jorge.View.MainPage">
<ContentPage.BindingContext>
<vm:MainPageViewModel />
</ContentPage.BindingContext>
<VerticalStackLayout Padding="20" Spacing="10">
<Label Text="Lista de tareas"
FontSize="24"
HorizontalOptions="Center" />
<Button Text="Añadir tarea"
Command="{Binding AddItemCommandNewWindow}" />
<CollectionView ItemsSource="{Binding Items}">
<CollectionView.ItemTemplate>
<DataTemplate>
<StackLayout Orientation="Horizontal" Padding="10">
<CheckBox IsChecked="{Binding IsCompleted}" />
<Label VerticalOptions="Center">
<Label.FormattedText>
<FormattedString>
<Span Text="{Binding Title}" />
</FormattedString>
</Label.FormattedText>
<Label.Triggers>
<DataTrigger TargetType="Label" Binding="{Binding IsCompleted}" Value="True">
<Setter Property="TextDecorations" Value="Strikethrough" />
</DataTrigger>
</Label.Triggers>
</Label>
<Button Text="Eliminar"
BackgroundColor="Red"
TextColor="White"
Command="{Binding Source={RelativeSource AncestorType={x:Type vm:MainPageViewModel}}, Path=DeleteItemCommand}"
CommandParameter="{Binding .}" />
</StackLayout>
</DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>
</VerticalStackLayout>
</ContentPage>
```

Este código define el AddItemNewWindowViewModel, que es el ViewModel responsable de gestionar la lógica para agregar una nueva tarea. La clase está marcada con el atributo QueryProperty para recibir los datos de la página anterior a través de una propiedad llamada SerializedItems, que contiene una cadena JSON con las tareas existentes. Cuando esta propiedad se establece, la cadena JSON se deserializa en una lista de tareas (ObservableCollection<TodoItem>), que es lo que se usará para mostrar y manipular las tareas en la interfaz.

Dentro del ViewModel, tengo dos propiedades principales: NewTaskTitle y NewTaskIsCompleted, que están vinculadas a los controles en la interfaz de usuario

(una entrada de texto para el título y un CheckBox para indicar si la tarea está completada). Estas propiedades permiten que el usuario ingrese el nombre de la tarea y marque si está completada o no.

También definí dos comandos: AddTaskCommand y CancelCommand. El comando AddTaskCommand se ejecuta cuando el usuario quiere agregar una nueva tarea. Dentro de este comando, se crea un nuevo objeto TodoItem con el título y el estado de completado proporcionado por el usuario. Luego, obtengo el MainPageViewModel usando DependencyService.Get<MainPageViewModel>() y agregué la nueva tarea a la colección de tareas en ese ViewModel mediante el método AddNewItem. Después de agregar la tarea, navego de vuelta a la página principal utilizando await Shell.Current.GoToAsync("///MainPage").

El comando CancelCommand permite al usuario cancelar la operación y volver a la página anterior, utilizando await Shell.Current.GoToAsync(".."), lo que simula una navegación hacia atrás.

En resumen, este ViewModel maneja la lógica para agregar una nueva tarea a la lista de tareas de la página principal y para cancelar la operación y volver a la página anterior.

```
using Newtonsoft.Json;
using System.Collections.ObjectModel;
using System.Windows.Input;
using ud04part1jorge.Models;

namespace ud04part1jorge.ViewModels
{
    [QueryProperty(nameof(SerializedItems), "pItems")]
    2 referencias
    public class AddItemNewWindowViewModel : BaseViewModel
    {
        private string newTaskTitle;
        1 referencia
        public string NewTaskTitle
        {
            get => newTaskTitle;
            set => SetProperty(ref newTaskTitle, value);
        }

        private bool newTaskIsCompleted;
        1 referencia
        public bool NewTaskIsCompleted
        {
            get => newTaskIsCompleted;
            set => SetProperty(ref newTaskIsCompleted, value);
        }

        private string serializedItems;
        1 referencia
        public string SerializedItems
        {
            get => serializedItems;
            set
            {
                serializedItems = value;

                if (!string.IsNullOrEmpty(serializedItems))
                {
                    Items = JsonConvert.DeserializeObject<ObservableCollection<TodoItem>>(serializedItems);
                }
            }
        }
    }
}
```

```

}
1 referencia
public ObservableCollection<TodoItem> Items { get; set; } = new ObservableCollection<TodoItem>();
1 referencia
public ICommand AddTaskCommand { get; }
1 referencia
public ICommand CancelCommand { get; }
0 referencias
public AddItemNewWindowViewModel()
{
    AddTaskCommand = new Command(async () =>
    {
        var newTask = new TodoItem
        {
            Title = NewTaskTitle,
            IsCompleted = NewTaskIsCompleted
        };
        var mainPageViewModel = DependencyService.Get<MainPageViewModel>();
        mainPageViewModel.AddNewItem(newTask);
        await Shell.Current.GoToAsync("///MainPage");
    });

    CancelCommand = new Command(async () =>
    {
        await Shell.Current.GoToAsync("..");
    });
}
}

```

Este código define una clase llamada BaseViewModel, que implementa la interfaz INotifyPropertyChanged. Esta interfaz es fundamental en el patrón modelo vista controlador (Model-View-ViewModel), ya que permite que la vista (UI) se actualice automáticamente cuando una propiedad en el ViewModel cambia.

Dentro de esta clase, tengo un evento llamado PropertyChanged, que es el que se dispara cuando alguna propiedad del ViewModel cambia. La función principal que tengo aquí es SetProperty<T>, que me permite establecer el valor de una propiedad y, si ese valor cambia, notificar a la vista para que se actualice. Para hacer esto, la función compara el valor actual con el nuevo valor; si son diferentes, se actualiza la propiedad y se llama al método OnPropertyChanged, que es el encargado de disparar el evento PropertyChanged.

El atributo CallerMemberName en el parámetro propertyName permite que, al llamar a SetProperty, se pase automáticamente el nombre de la propiedad que está siendo modificada, lo que facilita el proceso y elimina la necesidad de pasar el nombre de la propiedad manualmente.

En resumen, esta clase es una base para cualquier ViewModel en mi aplicación que necesite notificar a la vista cuando una propiedad cambia, lo que permite que los controles en la interfaz de usuario se actualicen de forma automática.


```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace ud04part1jorge.ViewModels
{
    2 referencias
    public class BaseViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        2 referencias
        protected void SetProperty<T>(ref T backingStore, T value, [CallerMemberName] string propertyName = "")
        {
            if (EqualityComparer<T>.Default.Equals(backingStore, value))
                return;

            backingStore = value;
            OnPropertyChanged(propertyName);
        }

        1 referencia
        protected void OnPropertyChanged([CallerMemberName] string propertyName = "")
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}

```

En este código, tengo la clase MainPageViewModel, que es responsable de gestionar la lista de tareas y las interacciones con la interfaz de usuario de la página principal. Primero, defino una colección observable de elementos TodoItem, llamada Items, que contiene dos tareas iniciales con títulos y estados de completado.

Luego, defino dos comandos: AddItemCommandNewWindow y DeleteItemCommand. El comando AddItemCommandNewWindow se encarga de serializar la lista de tareas Items en formato JSON y luego navega a la página AddItemNewWindow pasándole la lista de tareas serializada como parámetro. Esto permite que la página de añadir nueva tarea pueda acceder a la lista de tareas actuales.

El comando DeleteItemCommand se usa para eliminar una tarea de la lista cuando se invoca, verificando si la tarea está presente en la colección Items y, si es así, eliminándola.

Finalmente, la función AddNewItem me permite agregar una nueva tarea a la colección Items desde otro ViewModel (como en el caso de AddItemNewWindowViewModel), lo que permite actualizar la lista de tareas sin necesidad de acceder directamente a MainPageViewModel desde la vista.

Este ViewModel sigue el patrón modelo vista controlador donde la lógica de negocio (agregar o eliminar tareas) se maneja aquí y se vincula a la interfaz de usuario a través de la propiedad Items y los comandos.

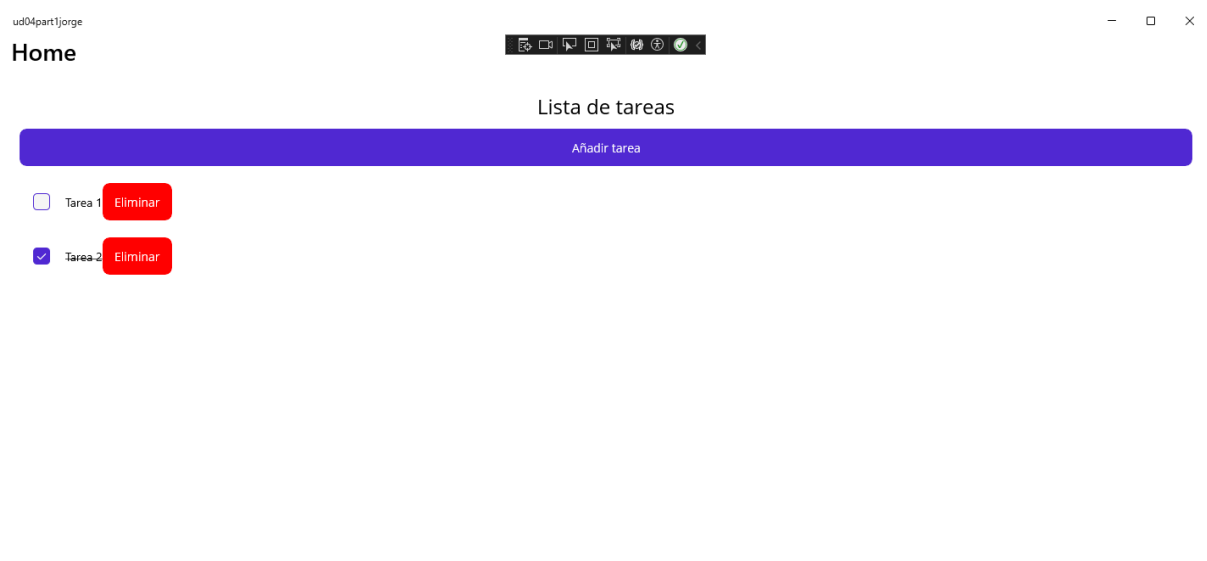
```

using ud04part1jorge.Models;

namespace ud04part1jorge.ViewModels
{
    3 referencias
    public class MainPageViewModel : BaseViewModel
    {
        5 referencias
        public ObservableCollection<TodoItem> Items { get; set; }
        1 referencia
        public ICommand AddItemCommandNewWindow { get; }
        1 referencia
        public ICommand DeleteItemCommand { get; }
        0 referencias
        public MainPageViewModel()
        {
            Items = new ObservableCollection<TodoItem>
            {
                new TodoItem { Title = "Tarea 1", IsCompleted = false },
                new TodoItem { Title = "Tarea 2", IsCompleted = true }
            };
            AddItemCommandNewWindow = new Command(async () =>
            {
                string serializedItems = JsonConvert.SerializeObject(Items);
                await Shell.Current.GoToAsync($"///AddItemNewWindow?pItems={Uri.EscapeDataString(serializedItems)}");
            });
            DeleteItemCommand = new Command<TodoItem>(item =>
            {
                if (Items.Contains(item))
                    Items.Remove(item);
            });
        }
        1 referencia
        public void AddNewItem(TodoItem newItem)
        {
            Items.Add(newItem);
        }
    }
}

```

La view de MainPage



La view de AddItemNewWindow

Add Item



Nueva tarea

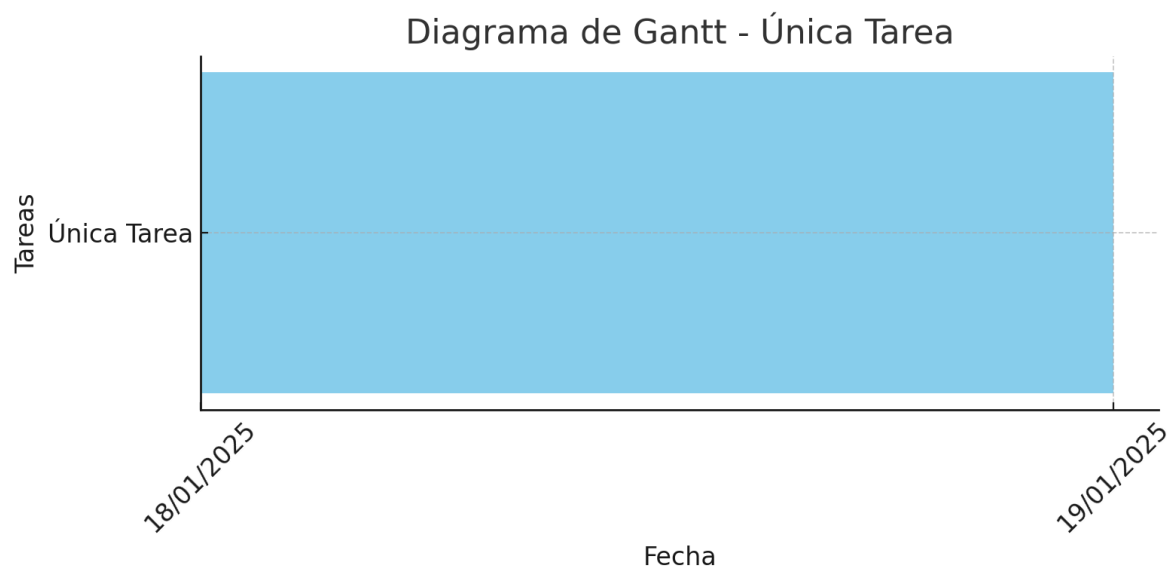


¿Está completada?

Añadir

Cancelar

Diagrama de Gantt

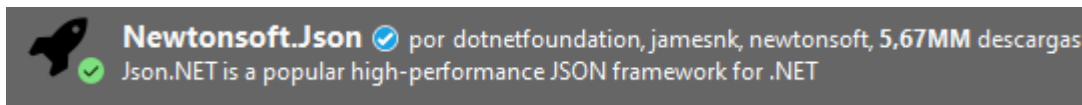


Problemas

A la hora de realizar este trabajo me he encontrado varios problemas. Algunos los he podido resolver y otros no. El `Shell.Navigation.GotoAsync()` porque este me daba problemas, así que utilice el `await Shell.Current.GoToAsync()`; en vez del anterior, para ello tuve que configurar el `AppShell.xaml` para que funcionara.

Tengo problemas con el `mainPageViewModel.AddNewItem(newTask)`; en la clase `AddItemNewWindowViewModel`. Me da problemas con el objeto no está instanciado en `Object`, o eso creo entender.

A la hora de transferir la información de `MainPageViewModel` a `AddItemNewWindowViewModel` me surgían errores, y una solución que creo que he encontrado es serializar la información; reestructurar la información en una forma mucho más fácil de almacenar. Aunque, debido a otros problemas, no sé si lo habré solucionado porque la aplicación no funciona. Respecto a serializar la información, me tuve que descargar el paquete:



Debido a que habían problemas que no sabía resolver consulte chatGPT y me recomendo crear la clase: `BaseViewModel`.

Sugerencias

No tengo ninguna.

Conclusión

Dejo constancia en este apartado del documento que he utilizado la herramienta de chat GPT para que me ayudase en algunos aspectos del proyecto.

Pese al resultado, este proyecto es un ejemplo de lo importante que es aprender a manejar la estructura modelo vista controlador. Debido a su importancia, continuaré trabajando en otras asignaturas y proyectos personales.