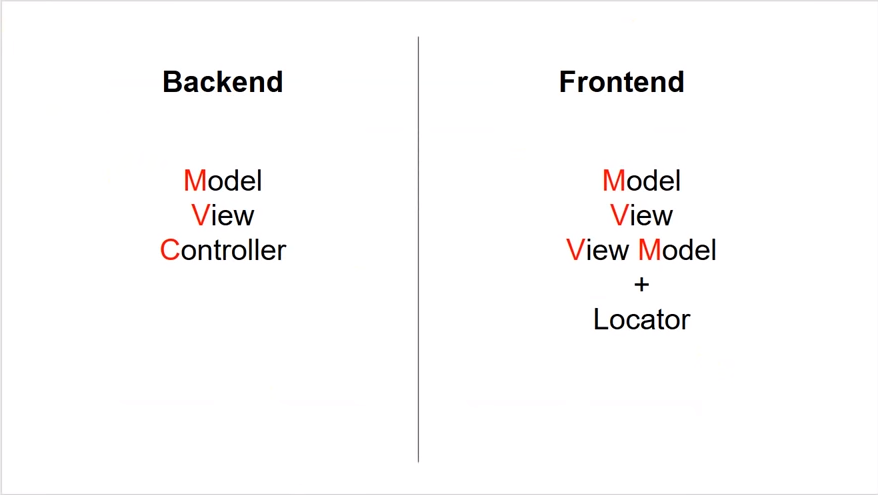


Android, iOS y UWP utilizan el proyecto “Sales”. Hay una “dependencia” de Sales.

No hay ninguna flecha apuntando a Common, por lo tanto Common no depende de nadie.

Domain usa a Common.

Backend usa a Domain y a Common



Para el Backend, nos va bien utilizar el patrón MVC, mientras que para el Frontend, nos va mejor utilizar el patrón MVVM.

MVVM: Patrón de arquitectura basado en Bindings.

Locator: Es otro patrón de arquitectura. No sirve para que, en cualquier momento, podamos encontrar la “MainViewModel” y tener una única “MainViewModel en toda la ejecución del programa. La MainViewmodel es una clase muy importante. Es la viewmodel principal.

El Locator sirve principalmente para poder ligar las “Viewmodels” con las “Views”.

MVC: Conectaremos los controladores con las vistas. Pero no se conectan a traves de Bindings sinó que le pasaremos los objetos. El Controlador le pasa el modelo a la vista. El usuario interactúa con la vista. Por ejemplo cuando yo voy a modificar un producto, quién es el modelo ?, El producto.!! . El controlador que hace ?, va y busca el producto en la base de datos. Lo tiene ahí en memoria. Y se lo pasa a la vista “Edit”, el usuario le cambia el precio y cuando le dice “guardar”, vuelve al controlador. El controlador valida que cumpla con las reglas de negocio y lo vuelca a la base de datos.

MVVM: Es muy diferente. El ViewModel tiene el producto. El Viewmodel dice “Yo soy producto” y la vista le dice “yo estoy ligado con vos”. Si realizo un cambio en la vista, se debe reflejar en la ViewModel.

Empezamos por el Backend. Vamos a hacer un “Catálogo de productos”. Quién sería el “modelo” ? El Producto.!. El producto representa al objeto que vamos a manipular. Producto será una clase que utilizaremos en “todo” el proyecto. En el backend, en la api, en android, etc..

Y, como se trata de una clase que la vamos a usar en todo el proyecto, en dónde debería estar ? En “Common”.!!

Servicios Web:

Cada vez que vayamos a consumir un servicio, necesitamos especificar 3 elementos :

Por ejemplo, para hacer un GET del servicio https://salesapiservices.azurewebsites.net/api/Products

Necesitamos especificar: 1) La URL BASE, 2)El Prefijo y 3) El Controlador

La URL Base sería <https://salesapiservices.azurewebsites.net>,

El prefijo sería: **/api**

Y el controlador sería: **Products**

Cuando yo tengo un string json y queremos convertirlo a objeto, tenemos que hacer una operación que llamamos “deserializar”.

Cuando convertimos un objeto a string se llama “Serializar”

MVVM + Locator

Para implementar el patrón MVVM con el patrón “Locator”, vamos a crear la carpeta Infrastructure y dentro de la carpeta creamos una clase llamada “InstanceLocator”.

El único objetivo de esta clase es instanciar la MainViewModel.

Entonces: El proyecto arranca por App.xaml.cs, al pasar por InitializedComponent inicializa el diccionario de recursos. Luego indica que la pagina principal será ProductPage, entonces, sabe que tiene que pintar la ProductPage. Esta pagina esta “bindada” a un objeto llamado “Main”. De donde obtiene Main ? y la encuentra con una llave llamada “Locator” que se encuentra dentro del diccionario de recursos.

Entonces se va para el diccionario de recursos (que está dentro de la App.xaml) y busca la llave “Locator” y ve que el Locator es una “instancia” del “InstanceLocator”. Y a este InstanceLocator lo encuentro en Infra, y puede ver que infra es una clase que se encuentra en Sales.InfraStructure , entonces se va a Sales.Infrastructure y encuentra que “Main” es una nueva instancia de la MainViewmodel. Todo esto para que la la ViewModel de Productos quede ligada a la MainViewModel.

Ahora hay que tener en cuenta otra cosa. Los “binding” manejan “sub-binding”.

No es cuestión de tirarle todo el código a la MainViewModel, por eso se recomienda utilizar “sub-binding”. Y el sub-binding se recomienda crear por pagina. Por ejemplo, si existe una view llamada “ProductPage”, debe existir una ViewModel llamada “ProductViewModel”, Casi siempre las “pages” y las “Viewmodel” van en parejitas. Hay veces que no es posible pero hay que tratar de mantener las parejistas.

La MainViewModel nos sirve para “controlar” todas las demas ViewModels

Al contenedor principal de la view ProductsPage.xaml, yo le puedo crear la propiedad BindingContext y bindarla a Products. Entonces la pagina tiene un doble binding. Esta bindada a Main, y está bindada a Products que es un “atributo” dentro de Main.

Hay una caracteristicas especial de las Viewmodels. Si yo hago un cambio en Products se debiera deslplegar en la ViewModel. Ese es el objetivo del patrón. Pero resulta que yo cambio la lista productos y no se me refleja de forma automática. Para lograr que se refleje en forma automatica hay que implementar una interfaz que es la InotifyPropertyChanged. Para poder encapsular toda esa propiedad vamos a agregar una clase al proyecto, la “BaseViewModel” que es la que implementa la interfaz InotifyPropertyChanged. Y tiene un método llamado “SetValue” que es un método generico que sirve para que me refresque cualquier campo.

Entonces la clace ProductsViewModel ca a heredad de la BaseViewModel. Y cada propiedad que yo quiero que me refresque vamos a crearle un atributo privado. Es decir, si yo tengo una propiedad publica llamada Products, vamos a crearle una atributo privado llamada products (p minúscula)

Cuando instanciamos en el App la ProductsPage, ésta instancia automáticamente a la MainViewModel. Cuando la MainViewmodel se instancia, también se instancia automaticamente la ProductsViewModel. Y cuando ProductViewModel se instanció, lo primero que hizo fue ejecutar el “LoadProducts” quien llamó al método “GetList” (de la ApiService)