Πίνακας περιεχομένων

[Section 1: Introduction 2](#_Toc152697658)

[Video 1: Introduction 2](#_Toc152697659)

[Video 2: What is .Net MAUI 2](#_Toc152697660)

[Video 3: Create first project 3](#_Toc152697661)

[Video 4: Project Structure 4](#_Toc152697662)

[Video 5: Three elements of stateful 7](#_Toc152697663)

[Video 6: Page, Layout and View, Namespaces. 8](#_Toc152697664)

[Video 8: URL Based navigation. 10](#_Toc152697665)

[Video 9: Basics of ListView and Data Binding. 12](#_Toc152697666)

[Video 10: Events Handling of ListView. 14](#_Toc152697667)

[Video 11: Parameters in URL based Navigation and static repository. 16](#_Toc152697668)

[Video 12: Stacklayout for Edit Contact page. 18](#_Toc152697669)

[Video 13: View Contact Details & Update Contact. 20](#_Toc152697670)

[Video 14: Observable Collection. 21](#_Toc152697671)

[Video 15: Field Validation. 22](#_Toc152697672)

Learning .Net MAUI while creating a **context** app in .Net 7

# Section 1: Introduction

## Video 1: Introduction

The technologies that it will cover are:

* **.Net MAUI,**
* **MVVM,**
* **Clean Architecture,**
* **SQLite,**
* **ASP .Net Core Minimal API.**

For the .Net Maui, it will use the **Event Driven Approach** first and then the **MVVM** to create the application.

It will also adopt the **Clean Architecture,** by usingthe use **case driven** and **plugin-based architecture**. Where it will rely on the need of **creating scalable and maintainable applications.**

**SQLite for persisting the data.**

Last but not least it will **work on creating and consuming ASP .Net Core Minimal API within .Net MAUI.**

*What the app will be:*

This is a contacts application.

It can add a contact, enter the name, email, phone number and so on.

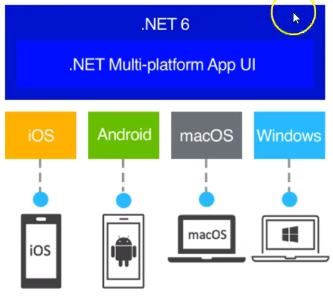
It can also delete the contact, edit a contact, update it, and also it can search contacts.

And it has field validation functionalities.

## Video 2: What is .Net MAUI

**What is .Net MAUI** and what is **the difference between .Net MAUI and Xamarin Forms.**

* The documentation of Microsoft shows that .Net MAUI allows us to create one code base, but targeting different type of platforms.
* That means we don’t have to write different code for different platforms.
* That code base is in one place and that code can be deployed in different platforms.
* Also, no need to make specific changes for the specific target platform.



Code Base 🡪

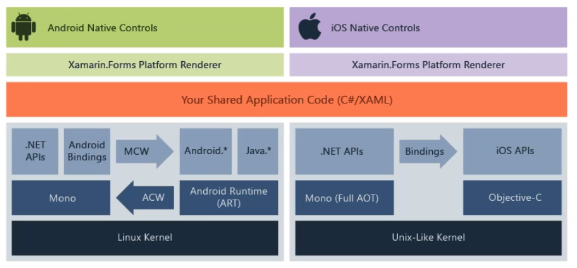
Type of platforms🡪

But isn’t this the same as the Xamarin forms?

In .Net MAUI once we create the project in the Visual Studio there is only **one Project.**

On the other hand, in Xamarin Forms, for different platforms (Android, IOS), there are different projects.

Android Project IOS Project



But Xamarin also has a shared code base, and that itself is another project. So, by creating Xamarin Forms, there are different projects. The shared application code contains all of the logic and the forms.

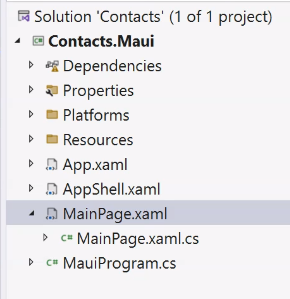
So .Net MAUI and Xamarin Forms are kind of similar but very different.

## Video 3: Create first project

In the ‘Create a new project’ window there are 3 different project types, which are:

1. .Net MAUI App
2. .Net MAUI Blazor App
3. .Net MAUI Class Library

We will choose the .Net MAUI App. Bellow is the project’s layout

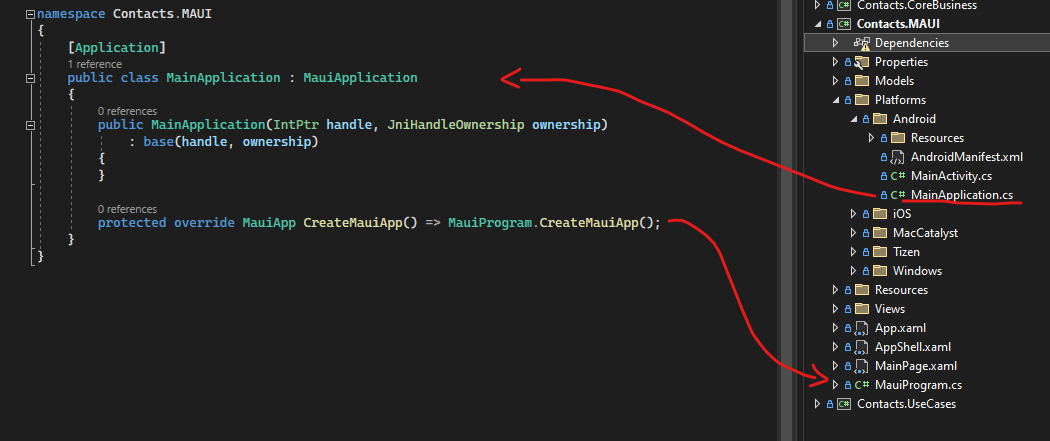


## Video 4: Project Structure

It seems like the MauiProgram.cs is the starting point of the project but is not. This application can be deployed in different type devices. So, how can this one program can be used in different platforms?

They must have their specific way to launch an application. The starting point is in the platforms folder.

* So, in the Android folder there is the MainApplication.cs and then this .cs file uses the MauiProgram.cs.



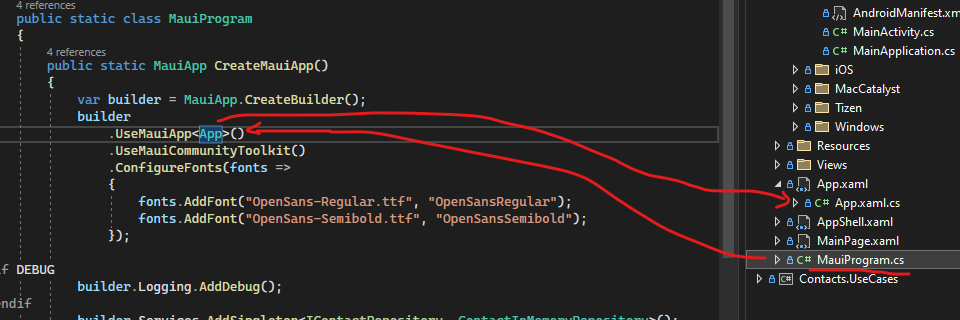
* IOS has a different way to launch the app. In the Program.cs which is the main app, it uses the AppDelegate.cs. And in the AppDelegate.cs there is a similar way of launching the MauiProgram.CreateMauiApp() method.
* Mac is similar to IOS.

The main point here is that every single platform points to its MauiProgram.cs.

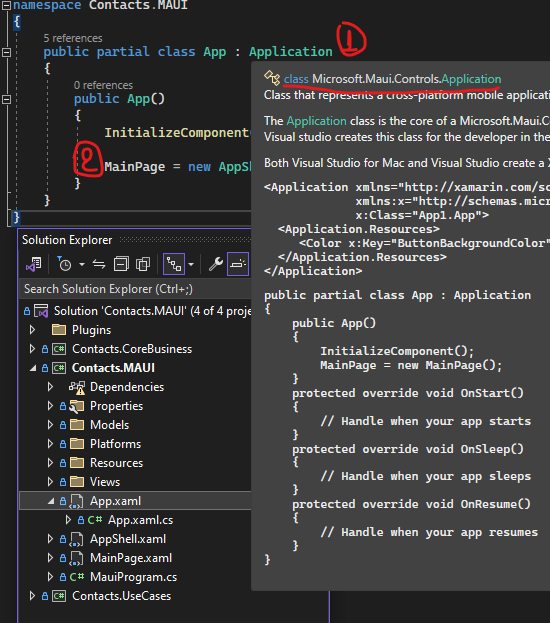
So, the main work is not in the platform folder. Only in the case that we want to write some code for a specific platform it goes there.

Practically everything starts form the CreateMauiApp() method which will be invoked from different platforms.

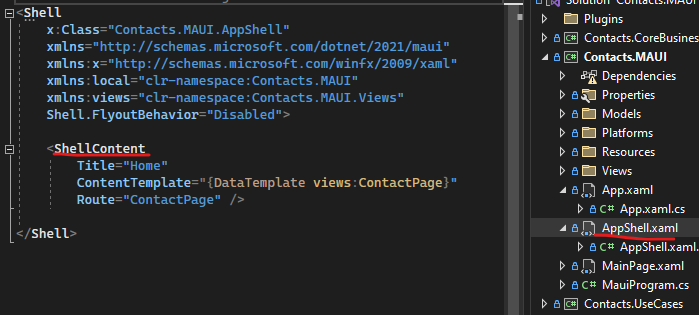
The next place to look at is the app.xaml.cs



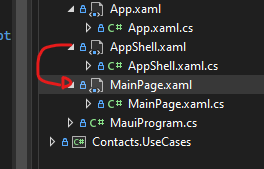
The frontend (App.xaml) focuses on resources coming from the Application class. *(Image 1 below)*



The App.xaml.cs has the MainPage property *(image 2 above)*, every single application has a MainPage property and this is similar with the Xamarin Forms. But in here it has a special type of MainPage, which is called the AppShell. So, in .Net MAUI a Shell application is created by default, we don’t need to follow this approximation but this option is provided by default. In other words, the Shell becomes the MainPage.

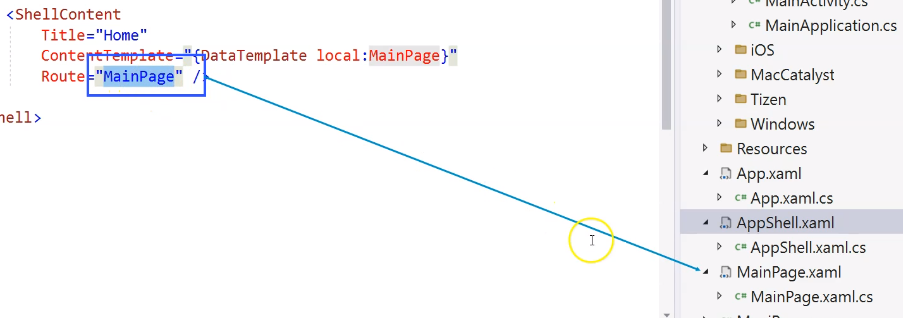


By going to the AppShell.xaml file (above), it has the <ShellContent> and if we go back to its code there is nothing on it. So, that means, the Shell itself is an opinionated way for organizing all of our pages. By default, the Route points in the MainPage (below) which MainPage is the default page of the .Net MAUI program.



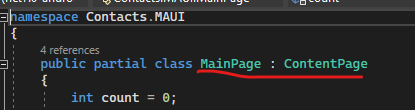
\*\* we don’t have to call it ‘main page’, it is just a page.

One think that is confusing and needs to clear up is that:

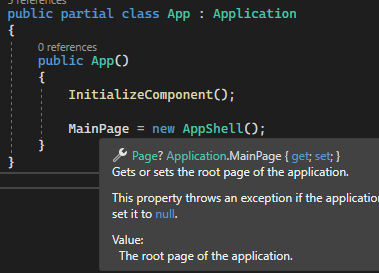


The MainPage in the route, refers to the MainPage.xaml.cs, which is just a page.

We can clear this up, also by looking to the MainPage.xaml.cs code, this MainPage Class is a ContentPage type:

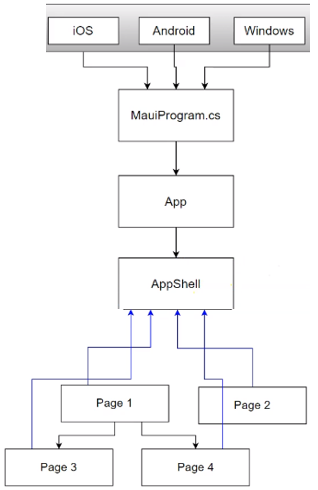


Whereas if we go to the App.xaml.cs code, the property MainPage is the Application’s MainPage, is a member of the application and the type of that is a page (below).



\*\*These are completely different objects.

*(Later, on the development of the app we change the name of the ‘MainPage’ page so it is clear).*

Now let’s clear up the flow and structure:

We have **platforms** (android, IOS, windows, etc.) and all these go to the **MauiProgram.cs.**

Then it goes to the **App**, App represents our application and **every application needs to have a main page.**

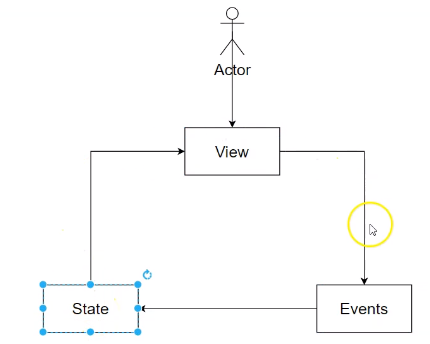
In .Net Maui the default MainPage is the shell page, which is the **AppShell**.

This organizes all the other pages.

No matter of what hierarchy the pages have, each of them need to be registered in the AppShell, for routing purposes.

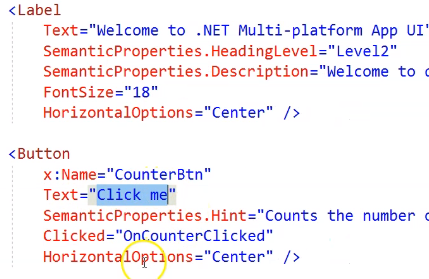
## Video 5: Three elements of stateful

**.Net MAUI application is a stateful application that has a user interface.** That means it needs to maintain a state, and a state is representing the data. There are 3 basic elements: one is the **state** which is the data, then the **view** represents the use interface, the state needs to be displayed in the view. Then the user will interact with the view and that generates the **events,** and then the developer will handle the events, which will change the state. It also means that the cycle goes on, and when the data is changed, the view is updated and the user interacts with the view and triggers the events.



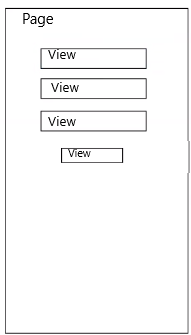
Now, in the default code that is generated when we create a .Net Maui app, it will show these 3 elements:

1. In the MainPage.cs class the state that represents the data is the *int count* and as we know the state is not initially displayed on the user interface.
2. The user interface is represented in the MainPage.xaml file.
3. The initial data is the text. Once the user clicks on the button,
4. Triggers the event and it handled by the event handler namely *OnCounterClicked*
5. Then the state is changed, *count++*
6. And the state is manually assigned to the text property of the button.
7. So that displays the state on the user interface, and complete the cycle.



This is the event driven mode (or State-view-event).

## Video 6: Page, Layout and View, Namespaces.



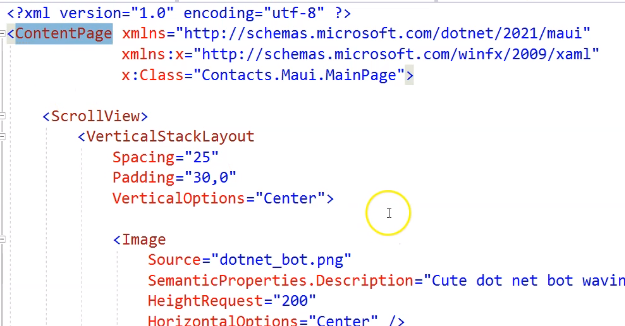
The View is the UI element. So, the application may have different pages. What is really on a page? The main page has the .xaml file and a .xaml.cs file. **The XAML file represents the UI** and **the .cs file is the code behind** which **provides logic and feeds data to be presented in the UI.**

**The UI part is a ContentPage**, in .Net MAUI may have different pages. It's basically just referring to that area that occupies the whole user interface. And then **inside the page, there could be different UI elements** like buttons, images, list views, and whatnot. So, if we put them in a diagram to express this more clearly, we have a page, and then inside the page, we may have different UI elements.

Under other development frameworks like web development frameworks, the UI elements are called controls. But **in the .Net MAUI,** **these are called UI element Views**.

\*\*We need to clear up something here. In the previous SVE diagram, the View is a generic term that refers to UI, and here these elements are called Views, each one of the elements.

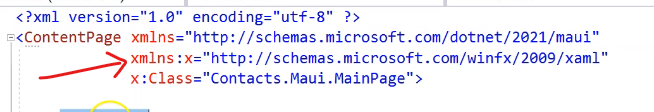
Each of these Views (or controls), are under from a layout. For example, <VerticalStackLayout> and <ScrollView> are layouts.

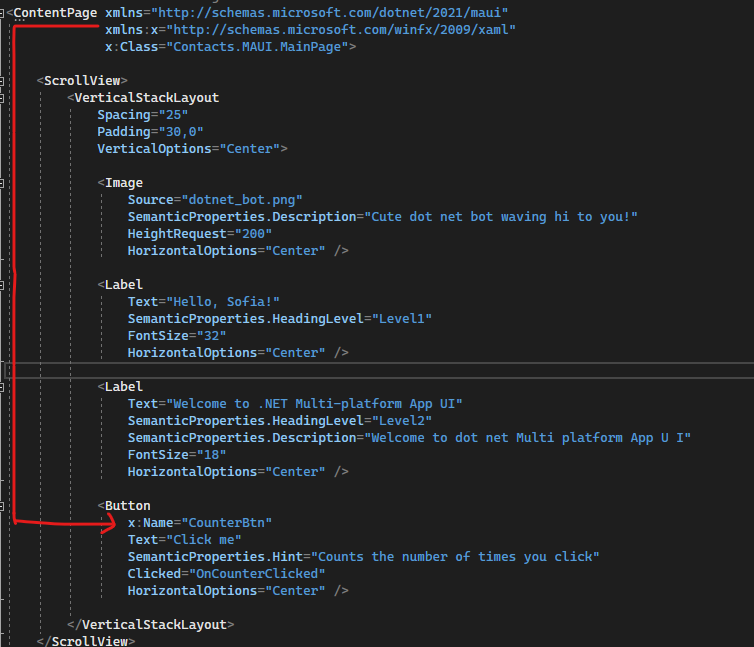
Of course, a layout is under a page, in this example is under the <ContextPage>

In other words, this is the hierarchy the .xaml file should follows.

Views 🡨 Layout 🡨 Page.

Another important factor to see in the .xaml file is the namespaces.



It provides these namespaces for the same reason as we do in C#, we provide definition for our keyword that we use inside a programming language. So, like any other language, here in order to get these VerticalStackLayout, image, label etc. keywords we need to specify the namespaces. For example the xaml namespace need to be specified like this: 

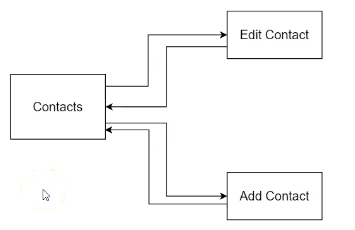
If we remove the ‘x:’ it will not regognize the Name keyword.

Also, if we remove the *x:Class=”Contacts.MAUI.MainPge”*  namespace, where this is the reference for the MainPage, the button’s Click event, will not have a definition.

I mention here the namespaces, because later on the development of our application, we will need to add namespaces manually, otherwise it won’t work because it is not a recognized keyword.

## Video 8: URL Based navigation.

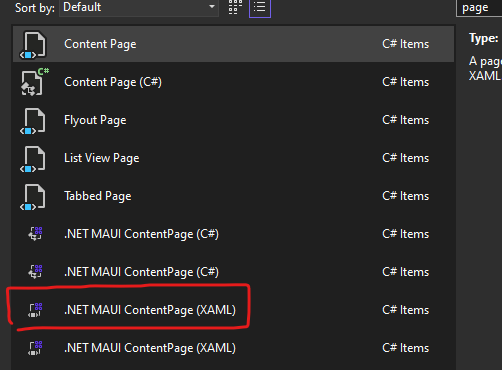
This video starts the implementation of the application, and it explains the below flow:

These are 3 pages in this context app.

There will be a Main context Page, and it is going to show a list of contacts, and then we will be able to navigate in the edit contact page and edit it and then navigate back. The other page will be to add a new contact and also navigate back.

In Visual Studio:

We start by add a new folder and name it Views. Then ‘Add New Item’ where if we enter page in the search, we can see some options and we will choose this one:

The .Net MAUI ContextPage(XAML), and not just the (C#). The C# only means that we will not have a xaml file, so that indicates that the front-end and back-end are written in the C# only. For this reason, to have everything separately and we structured we choose the XAML option.

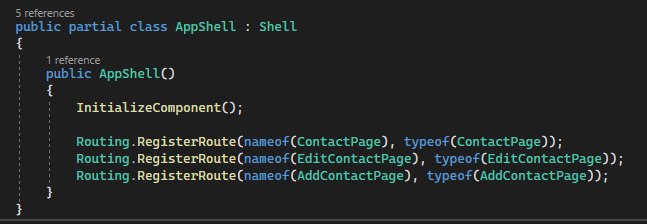
And we name it: ContactPage.xaml

And then in the namespace there is the title where we can make changes. Then add one more Page namely: EditContactPage.xaml and another one: AddContactPage.xaml

If we run the application, we will not see our pages. So, by hitting F5 the application runs in debug mode. That triggers the hot reload, that means if there are changes in the in the front-end file, they are reloaded.

To continue, the Shell provides the URL Based Navigation, and in its constructor, we can register the routes in order to navigate in any page we want. In other words, we let the system know that we have these routes, the points to these pages.

We can do this like this:



The first parameter is the name(key) and the second is the type. The name can be anything we chose and this is the key of the route and it has to be unique, for this reason we can give the name of our context page. The second signature is the type and we can use the type of our page. So now the inside has a key and a value pair. Later when we navigate to this page it knows the type of the page and it will be able to construct dynamically on demand in the memory and then show the page. That is because it knows the key value pair and also which type to instantiate.

So now let’s demonstrate how we can navigate to these 3 pages. Firstly, we don’t want the MainPage to be the home page, and in its place, we want to put our contact page. To do that we go to the front-end of the AppShell.xaml and in the Route we enter our page: Route=”ContactPage”. It also has a local where it comes from a namespace, it is an extended namespace where you can say local and then see what is defined under that namespace.

But our ContextPage is not under the local, for this reason we will add our namespace so it finds our page. To do that we can see the namespace in the ContactsPage.xmal.cs file and create a new one here, like the pic on the left. Then we can use views to provide our page.

So, if we run the app, it will display our ContactPage. Then if we add 2 buttons (for demonstration purpose), in order to show navigation via pages, like this in the <VerticalStackLayout>:

*<Button x:Name:”btnEditContact” Text=”Edit Contact”></Button>*

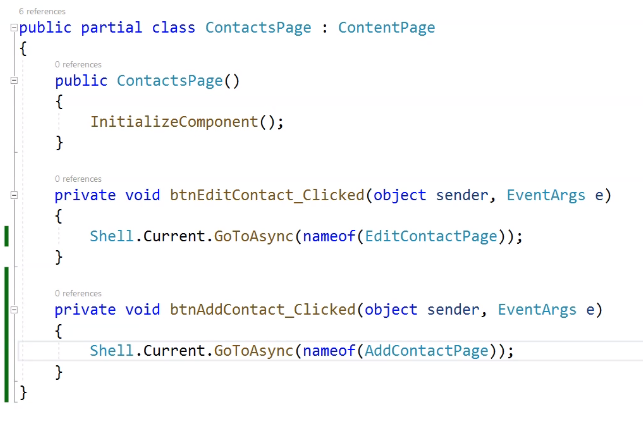
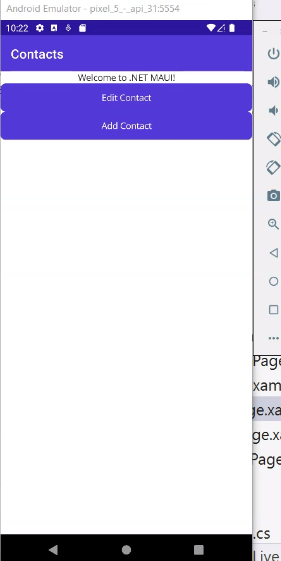
Then we add the event handler:

*<Button x:Name:”btnEditContact” Clicked=”btnEditContact\_Clicked” Text=”Edit Contact”></Button>*

For both edit and add page, then in the backend of this xaml we can see that it generated this event handler.

For the URL base navigation, we can write: Shell.Current.GoToAsync(nameof(EditContactpage))

The Current is a static member of the Shell class and it refers to the shell that we saw above, when we registered the routes, that means it knows all the registered routed and their paths. Also, in the string name paramenter in the GoToAsync method is the key of that particular route. It looks like this:

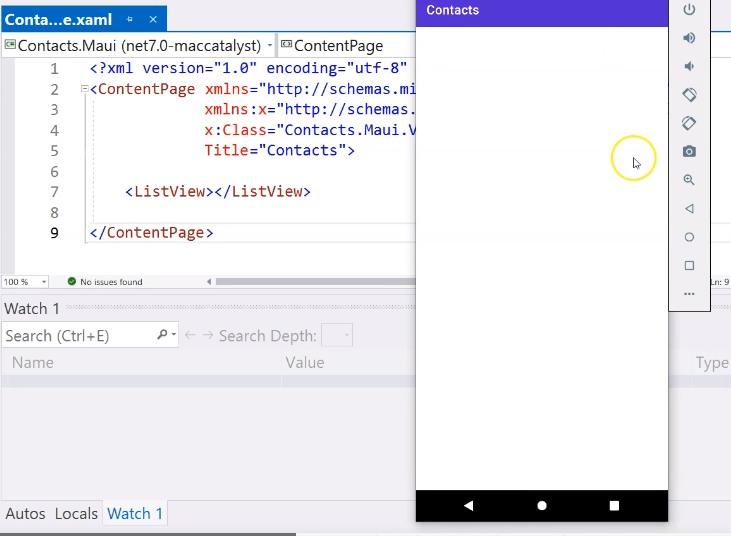
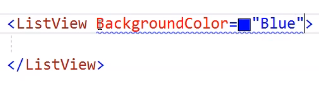
 

Then we can add a cancel button in both EditPage and AddPage and we can do the same Url Navigation. But it will not work properly and this is because the Contact page is the startup page of the app, and is not any other page that has a parent page. So we will not use the relative paths as before in the ContactPage.cs, instead we will use an absolute path that starts with ‘//’.



## Video 9: Basics of ListView and Data Binding.

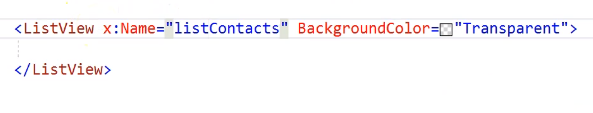
This video is showing how we can create a list of contacts and to do that we need a ListView. At the first videos we mentioned that can be only one View on the top level inside a page. So in the ContactPage we can see the VerticalStackLayout and everything else is inside it. So we delete all of these and add a <ListView> and if we run the app we can see that there is nothing on it.

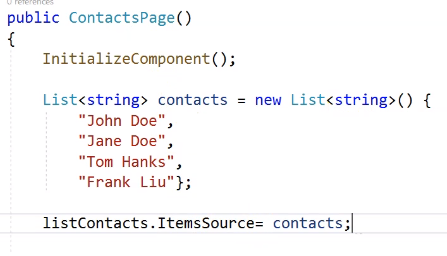
List View can have some parameters, for example BackgroudColor=”Blue” 

And the whole page became blue, that means that ListView occupies in the whole page.

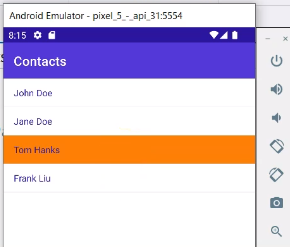
It also has the scroll functionality.

So, we want to show something in the ListView, we can add some dummy data, and then define it in the front end.

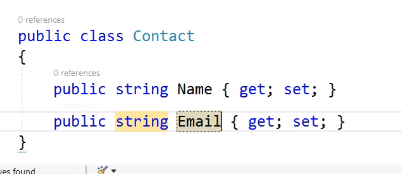
First, we give a name to the ListView and in the back-end we reference it to the list.



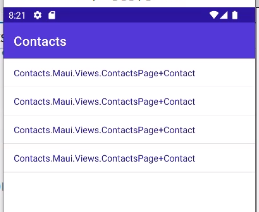
The ItemSource property is of type IEnumerable.

And this is the result:

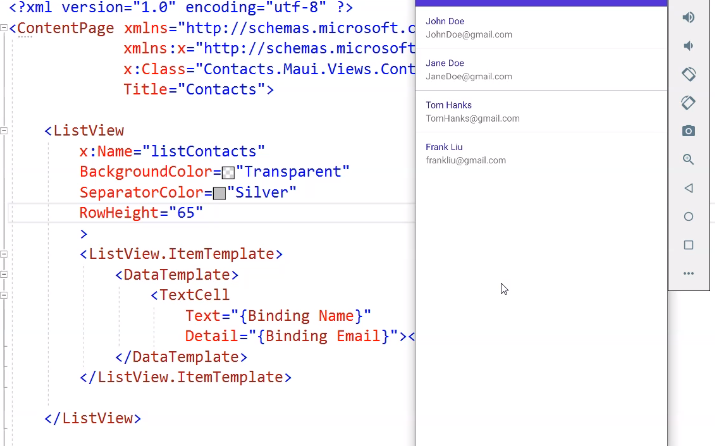
Now let’s see how to use data binding. For this example, we will delete the Contacts list and create a class for now in the same file with two properties, the name and email.

So now our class seems like this.

But the result is not as expected



In order to fix this, we need to tell the list view how to display the item. We can do this by using the ListView.ItemTemplate and DataTemplate in order to bind and display the items as we want:



The Binding means that we are referring to the object that is being bound to the item.

## Video 10: Events Handling of ListView.

In the previous video we mentioned how to map the **state (data)** with the **view (UI),** which is the basic data binding. In this video we will cover basic events for the ListView, that means we will do something with the ListView.

Usually, in our phone we have a list and we use our finger to tap on an item of that list. So, the events that we are interested in for now, called:

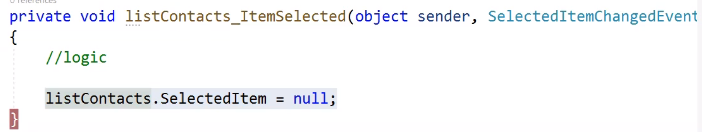
ItemSelected=”listContacts\_ItemSelected”

By accepting this event in the frond-end, it automatically generated it in the back-end.

If we debug this event in the back-end, we see that when we click on an item, the event is triggers, but when we click on the same item, multiple times, it does not trigger.

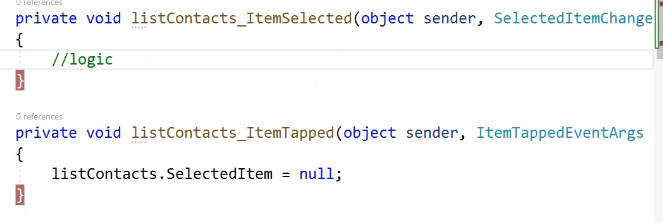
That means that the ItemSelected event handler is triggered when the item selection has changed. At the beginning there is not item selected.

So, one way to fix this problem, is to deselect the item within the event:



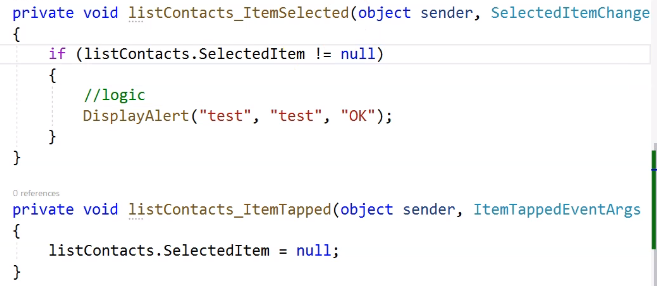
So now the item is deselected every time and it triggers again if we click the same.

Another event handler is the ”ItemTapped”. If we debug again, we can see that the first time the app runs, the ItemSelected event is triggered and if we continue, then the ItemTapped event is trigged right after. A more reliable way for deselected items is to do it in the ItemTapped event.

This is because the ItemTapped event is always trigged. When we select an item or not, whether we tap on the same item or in a different. So, this event is triggered way more times than the ItemSelected.

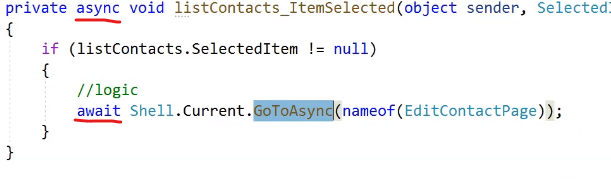
But here actually there is a small bug. The ItemSelected event is triggered twice, this is happening because, when we select an item, the item selected changed from null to something, and then because we deselect it is changed from something to null, and that triggers the event again.

There is a workaround to surpass this ‘bug’ and is this:

We add a validation that says, we are only dealing with anything that is actually a selected item, and if there is no selected item, we don’t do anything.

If we run the app again, we see that the alert only trigger once.

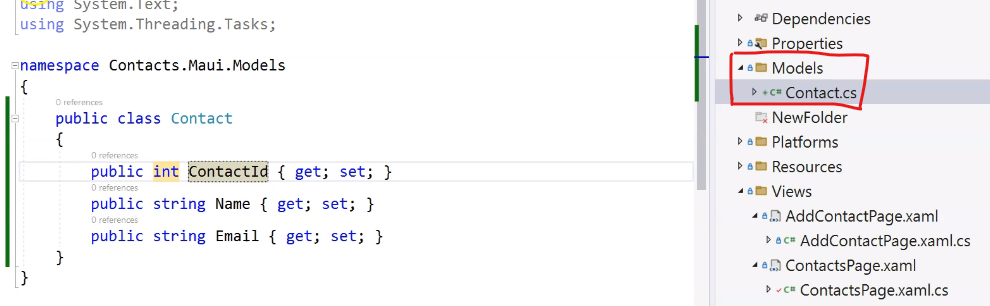
So, let’s put our actual logic in the event, and when we click on a contact we want to go to the EdiContactPage, and as showed in the previous video we do it like this:

and let’s make it to be asynchronous.

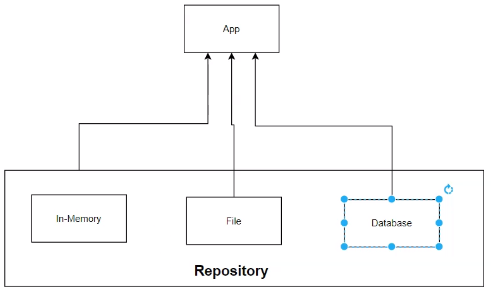
## Video 11: Parameters in URL based Navigation and static repository.

In this video we will create an Edit Contact Page to be able to see the details of that contact and edit it. Before we start, we need to clear up the code in the Contact page. For example, we need to separate the Contact Core class and also there is the hard coded list.

First, we create a folder namely Models and for now add the Contact class in there and add more properties.



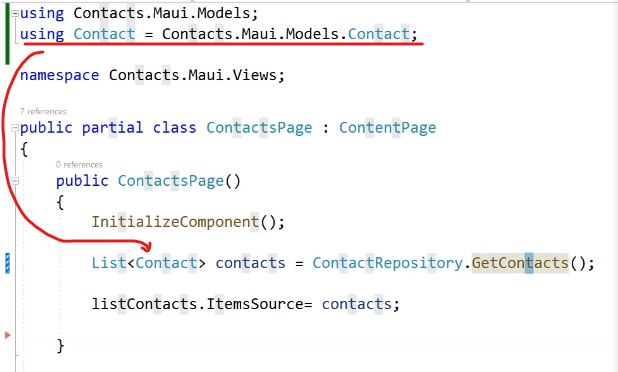
And now we go back to fix the Contact page class. Let’s clear some things here, every time we have to deal with data, even if this is hard coding data (in-memory data), or with a file to store the data, or with a database, we will need to retrieve this data, or apply the CRUD operations, so we need to have a central place to encapsulate the data operations. To achieve this, we can use repository. A repository is a central place where those operations are encapsulated, when we have to deal with different data from different sources.



For now, we will use a simple static repository. We will create a class under the *Models* folder, namely ContactRepository. This is a repository which serves the purpose of doing all those operations within the memory. Later when we have to deal with databases, we will use of course the clean architecture and different repositories for the operations, and dependency injection in order to inject those repositories into our application logic layer. So, for now we will have a static class and a static list of contacts. Apart for that we need to have a get method, for getting all the contacts, and a GetContactByID like this:

And call the GetContacts() method in the ContactPage constructor.

But we have to define from where we are getting the contact so in the namespace we reference this: *using Contact = Contacts.MAUI.Models.Contact;*



So now the next step is to click on a contact and go to the edit page. Firstly, we will show the details information of the contact and then we will deal with the update. To show the details we need to know which one we chose.

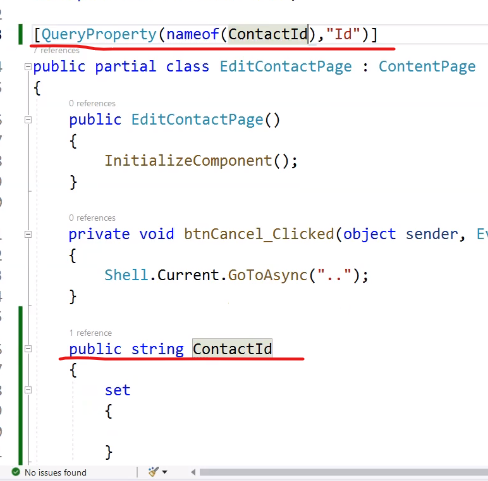
In our code, when a user clicks on a contact, we used the URL based navigation to go to the edit page, so we can use a query string, it is similar like wed app development. It will look like this:

To achieve this we going to use string interpolation, and inside the name of the page, then the question mark, and we will pass the Id. To get the Id we can use the listContact.ItemSected where this is an object and we can cast it into a contact in order to get the ContactId:



So, the first part is the query parameter and the second is the value.

Now in the Edit contact page, in order to receive the parameter value that is passed, we would need to have a property and that should correspond to that parameter value. This property will be the ContactId and it only needs a setter, where it will be triggered when the information is passed over. But how the property will know when the setter is triggered, for this we can use an annotation in the class.

The first parameter in the QueryProperty is the property name that we created in the code, and the second is the name of the query parameter from the contact page.

①

The query property (1) is telling us that whenever we receive the parameter value from the previous page, we will map that parameter value to this property (2).

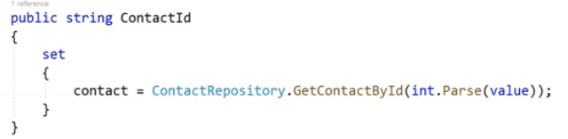
②

When that happens, the setter will be set and the value of that property is within the ‘value’ and this actually has the contact ID.

In order to add the value, we can call the method we created, the GetContactById() and pass the ‘value’ as the parameter and parse it into an integer. But if the value is inappropriate, we need to throw an exception, so we will store the result of it in private variable.

1. Add the namespace, so it finds the Contact.

 2) This is the private variable

3) This is how to set the setter.

 In order to test that this works, in the front end there is a label and we give a name to the label.



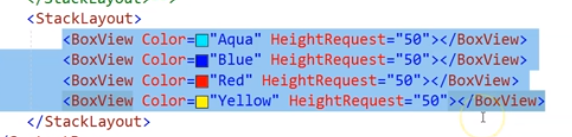
After that inside the set, we add this value to the label.

In this lesson we saw first, how to create a static repository (this is a mock data), then how to pass information from on page to another with the query parameter and then inside the place that we receive the information we used a write only or set property to pass this information.

## Video 12: Stacklayout for Edit Contact page.

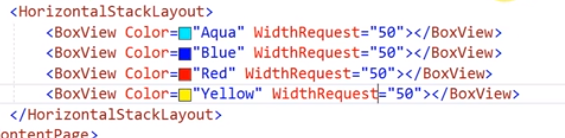
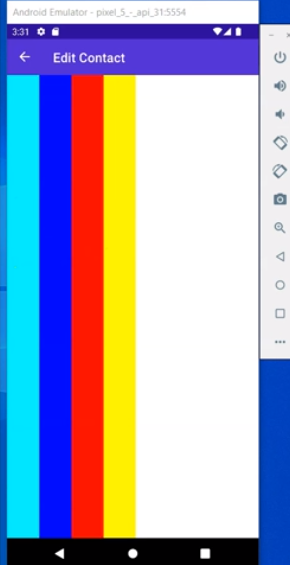
In this lesson we will familiarize ourselves with the stack layout. In .Net MAUI there is the <VerticalStackLayout> and the <HorizontalStackLayout>. The default orientation of the <StackLayout> is the <VerticalStackLayout>.

This is the <StackLayout> or the <VerticalStackLayout>.

If we use a BoxView we will have this:



So, this tells us that in the stack layout, each one of the child views are stacked together right up and down because the orientation is vertical. If I change the orientation (and the height request becomes, width request), then it will have this approximation:



Everything is stacked together again back it

stacked from the left to the right.

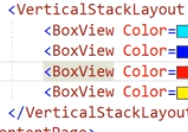
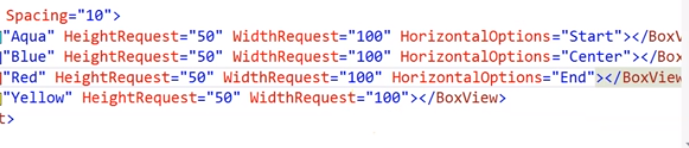
There is also a property with the name

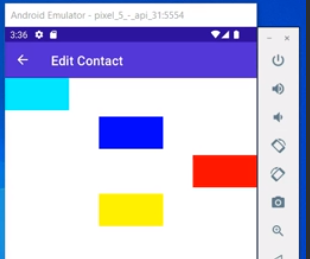
‘Spacing’ that we can define and this adds spaces

between the <BoxViews>. Another important

thing for Horizontal Stack Layout is that by default

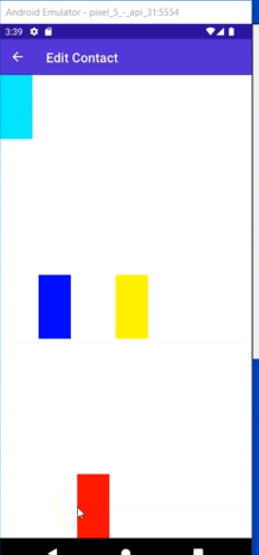
it fills the entire screen form bottom to top. And similarly for Vertical it fills the entire screen form the left to the right.





So, the above will show this:

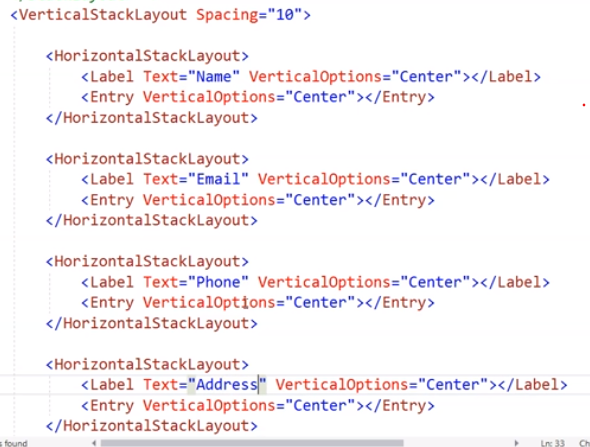
It will be the same for HorizontalStackLayout but we will have VerticalOptions instead of Horizontal an it will seem like this:



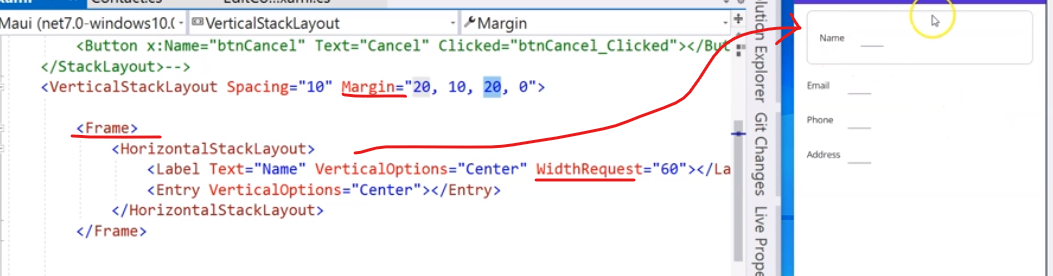
Another thing to know is that all of the layouts are nested. So, if we put a horizontal stack layout inside a stack layout this will be like this:



That all we need to know for the Stack Layouts, and to continue in the models folder in the Contact Class we will add 2 more properties, the Phone and Email. Now it is time to stop the experimental code in the EditContact.xaml and lets actually create it. So, first we need to add a VerticalStackLayout and inside to add a HorizontalStackLayout in order to add the child Views.

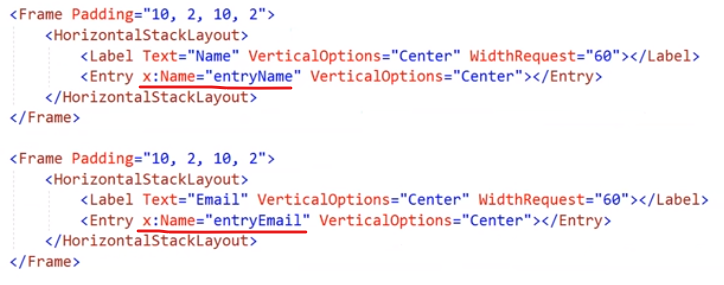


Now we adding some more stuff for beatifying the UI, like the margin, WidthRequest and frame and it becomes like this:



## Video 13: View Contact Details & Update Contact.

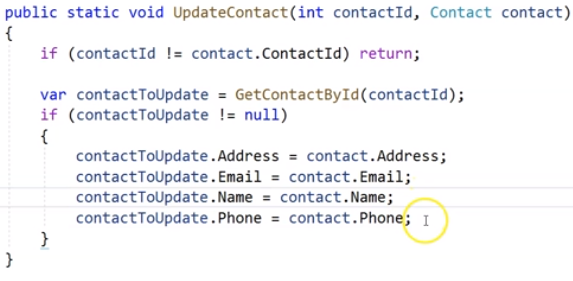
In this video, the edit contact page needs to display the contact information. To start with the EditContactPage.xaml we need to assign names in the <Entry>. We do this like:



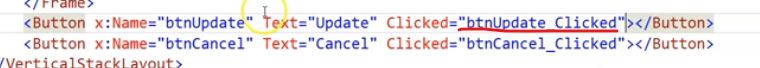
In the code behind, lets assign those values, but also to add a validation in order to be safe.

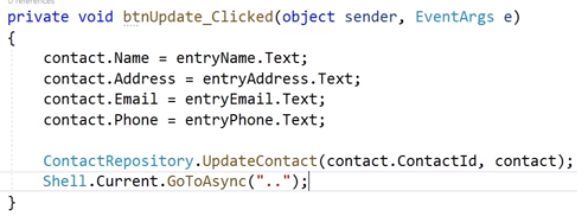


Now we will work on the update so in the ContractRepository.cs we will create a new method named UpdateContact:

So first of all, we will validate that the Contact ID passed in is the same as the ID in the contact object, if they are not then just return. Then we will use the GetContactById method and pass the result into a variable. When we will use a database, we will not follow this, but for now is good, so we will update all the fields except ID.

So, in the front end we will create a new button, the update and also it will generate the event in the back end.



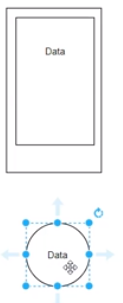
One of the reasons to use the Mvvm is to eliminate the process of manually populating the views, but for now will we do this. After the contact is updated then it needs to go back, and this is like clicking on the cancel button.

Everything works fine, when we make a change in the Edit page and then it goes back to the Contact page, we cannot see the change but, in the edit, contact we can see it.

## Video 14: Observable Collection.

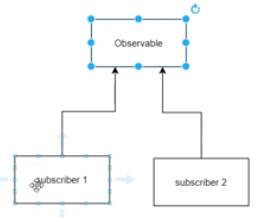
In the Contact page in the constructor, it gets the contacts from the repository and then add this list to the item source, and these lines of codes will be executed only once because it lives in the memory. So, there is place to call these 2 lines.

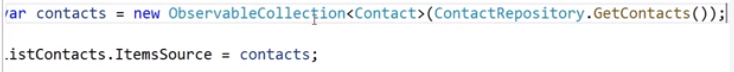
There is an event and we override it, this is the OnAppearing() and this is going to be called every time the page regains the focus. But it still did not work, and the problem is the second line with the ItemSource.

This is because, in .Net Maui there is a page that actually displays some data, and then we have some data that is loaded in the memory. when the list view is populated with the data, the list view stores a copy of that data in a different format, in the memory, but we have another copy of that data which is in the in the contact repository so, these two copies are not the same copy. In .Net Maui, if you don't notify the list view or any view that there is a change in the data, the data in the list view or any other views will not actually update. But you have to actually tell it to notify the view that there is a change in the data and then the view will do the update.

In memory

A way to notify the data is to use Observable data.

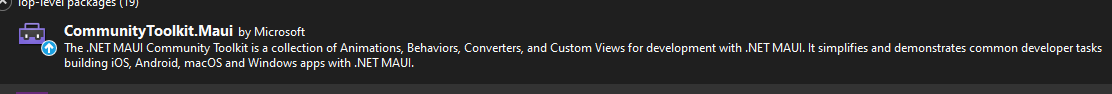
Let's say we only have two subscribers, and then the subscriber is actually subscribing to the observable data and when the observable data changes, it's going to actually send a notification. Because both Subscriber one and Subscriber two subscribe to the observable data, then it actually receives that notification and then the subscriber can do whatever it wants. In our case, our list view is trying to subscribe. But because we are not using observable data and we are using a list it's not working. We change the OnAppearing like this:



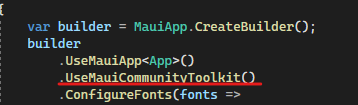
If we don’t do it like this it is just keeps the data in the memory without updating.

## Video 15: Field Validation.

This video shows how to do validation in the views of the Edit Contact page. For example, the name must be not null and the email must be a valid email. To achieve this, there is a NuGet package, the Community Toolkit, that helps.



In the MauiProgram.cs it will complain because we need to add to the builder this information:

This Toolkit contains different features but the ones that we interesting now is the validation behaviors.

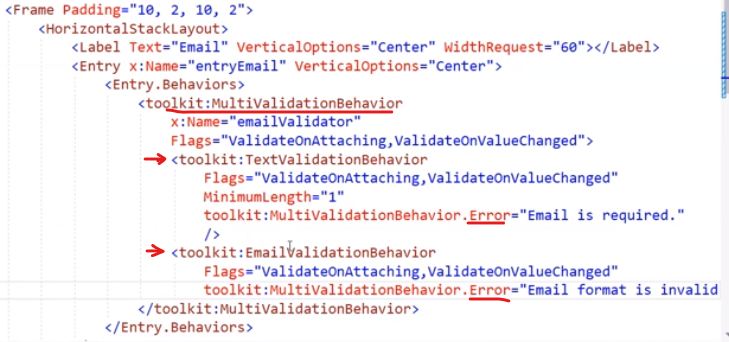
To add these behaviors, we first need to add the namespace.

and then:

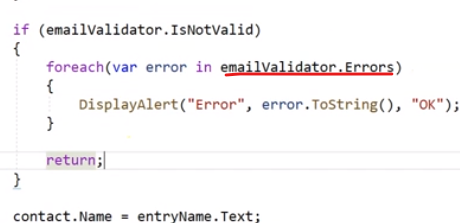
So, there is a validation in the minimum length to be one. Another property is the Flags. This configures the text behavior validation. Then we use a combination of OnAttaching and OnValueChanged. Of course, we give a name in order to reference the validation field, and then call it. And of course, there are more properties like the Regular expression in order to limit the format of the text.

In the backend now in order to check the name to have the value we do it like this:

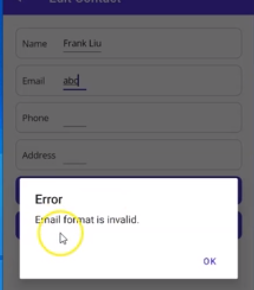
In the btnUpdate we specify that if the nameValidator is invalid then displayAlert.

There are more complex validations that can be applied like the MultiValidationBehavior where inside of it you can apply many validations, like in the image on the left. In the multiValidator we have the Textvalidation and the EmailValidation, and of course we can apply as many properties we want.

So, in the backend now in the btnUpdate again, we do it like this:

As we can see, it is possible to loop throw the error messages and also display them one by one.

If we run the program, and open a contact and then delete at all the email, if we try to update it will display 2 messages, the one will be that the Email format is invalid and the other one that the Email is required.

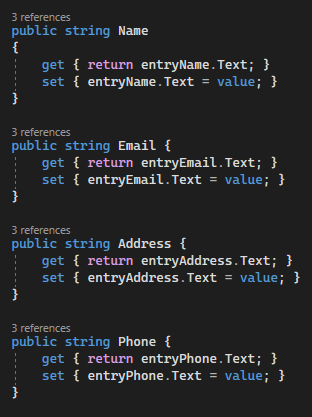
 If we put something like abc, then it will display this message.

## Video 15: Reusable Control.

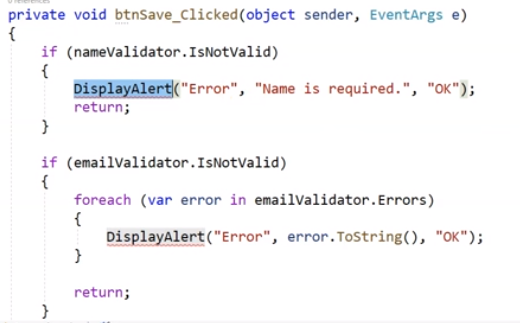
In this video we will create a reusable control, because the 2 pages (add and edit), are very similar but have different functionalities, so these 2 pages will share the same control.

In the Views folder -> create another folder, Controls -> create a file of type, .Net MAUI ContentView (XAML), namely ContactControl.xaml

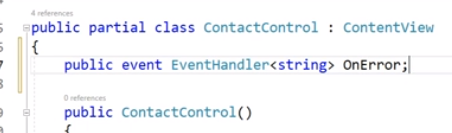
This is what it looks like when it is been generated. Then we go to the EditContact page, and cut the whole FE. Also, copy the toolkit namespace. All we want now is to share the Entry views, so we will share these out through properties. The properties will be in the back-end like this:

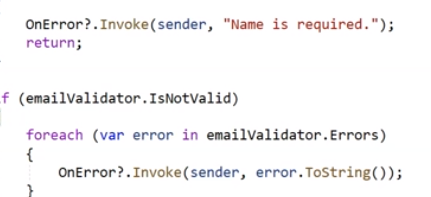
So, we actually get the text form the entries and then we set them. Now we should do the same for the buttons, by creating the event handlers as the previous videos.

After that we cut the code from the update first and paste it in out ContentView.



But we can see that the DisplayAlert is not available in here, and this is because it is only available in a Page (or ContentPage). So, we will replace the DiplaAlert with an event.

First, we create the Event. Then we invoke it.

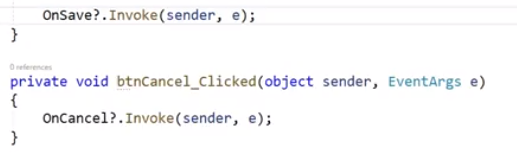


But how does the parent control know that the Save button is clicked?

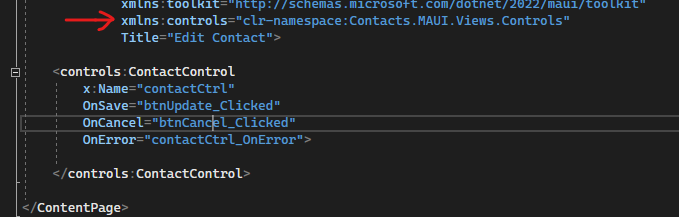
For this we will create some more events, for both save and cancel.



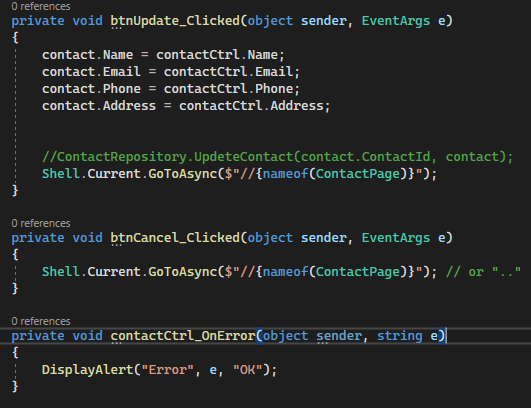
Then invoke them and notice the parent control that a button is clicked (For the save button all the validation passed and then invoked).



Now it is time to use it in the parent.

First define the ContentView in the namespace so it finds it. Then call the actual btnUpdate and btnCancel events, because we already have them, and finally create a new one for the onError.

Now, it is time to fix the code in the BE, and all we need to do is this:



Notice that we changed the instantiations, to be from the contactCtrl where our new props lives and not the entry views that now belong to the ContentView.

If we validate the app, it is working properly as before.