# List View and Table View Examples

## Creating a list type page and pages that navigate from it

We’ll create a content page that shows a list view with objects populating it. Tapping on the objects will open a custom content page that will show the object details. We will also create a class that inherits from ViewCell which will be used as a template for displaying the objects.

Change the MainPage call to display a Navigation Page passing in a new FruitListPage. Add a new Content Page (C#) called FruitListPage.cs Then create the content page definition:

public class FruitListPage : ContentPage

{

public FruitListPage()

{

List<Fruit> list = new List<Fruit> {

new Fruit { Name = "Apple", Description = "Awesome!" },

new Fruit { Name = "Banana", Description = "Beautiful!" },

new Fruit { Name = "Cherry", Description = "Cheap!" },

};

var listView = new ListView

{

ItemsSource = list,

ItemTemplate = new DataTemplate(typeof(FruitCell)), leave out initially

RowHeight = FruitCell.RowHeight,

};

listView.ItemTapped += (sender, e) => {

listView.SelectedItem = null;

Navigation.PushAsync(new FruitDetailPage(e.Item as Fruit));

};

Title = "Fruits";

Content = listView;

}

}

The fruit class is very simple, just a couple properties: (Note: we’ll come back to this later)

public class Fruit

{

private string fName = String.Empty;

private string fDesc = String.Empty;

public string Name

{

get { return this.fName; }

set { this.fName = value; }

}

public string Description

{

get { return this.fDesc; }

set { this.fDesc = value; }

}

}

Define a class that will be the display of a fruit object inside the list view (using Xamarin.Forms):

SetBinding: <https://docs.microsoft.com/en-us/dotnet/api/xamarin.forms.bindableobjectextensions.setbinding?view=xamarin-forms>

public class FruitCell : ViewCell

{

public const int RowHeight = 55;

public FruitCell()

{

// setting the binding will cause this label to look for a property

// called Name in the object this cell is linked to

var nameLabel = new Label { FontAttributes = FontAttributes.Bold };

nameLabel.SetBinding(Label.TextProperty, "Name");

var descriptionLabel = new Label { TextColor = Color.Gray };

descriptionLabel.SetBinding(Label.TextProperty, "Description");

// define what the cell will show. Usually start with a layout. Works the

Same as defining content for a content page in code.

View = new StackLayout

{

Spacing = 2,

Padding = 5,

Children = {

nameLabel,

descriptionLabel

},

};

}

}

Define a class that will be the content page when an item in the list is tapped, we setup the event handler for the tap earlier in the definition of the listview:

public class FruitDetailPage : ContentPage

{

public FruitDetailPage(Fruit fruit)

{

Title = fruit.Name;

Content = new Label

{

Text = fruit.Description,

HorizontalOptions = LayoutOptions.CenterAndExpand,

VerticalOptions = LayoutOptions.CenterAndExpand,

};

}

}

# More List Stuff (Add and Delete)

Adding the behavior to delete items as they are tapped. Comment out the Navigation stuff for now and replace with ((ObservableCollection<Fruit>)listView.ItemsSource).Remove((Fruit)e.Item);

Now run the app and tapping deletes the row tapped on.

Comment out the new code and uncomment the Navigation code.

Probably better to use the platform specific approach of a Context Action. A context action is defined in the view cell. Android and iOS behave differently in terms of how to access the content action list, in the end treat the action in the same way. In the FruitCell constructor add the following at the end:

ObservableCollection in Listview: <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/listview/data-and-databinding>

MenuItem mi = new MenuItem { Text = "Delete", IsDestructive = true };

mi.Clicked += (sender, e) =>

{

ListView parent = (ListView)this.Parent;

ObservableCollection<Fruit> list = (ObservableCollection<Fruit>) parent.ItemsSource;

list.Remove((Fruit)this.BindingContext);

};

ContextActions.Add(mi);

To get Add to work, create two Entries and a Button on the FruitListPage. Basically alter the assignment to Content to be this instead:

StackLayout layout = new StackLayout();

layout.Children.Add(listView);

Entry eName = new Entry();

Entry eDescription = new Entry();

Button btnNew = new Button { Text = "Create New Fruit" };

btnNew.Clicked += (sender, e)=>

{

Fruit f = new Fruit{Name=eName.Text,Description=eDescription.Text };

list.Add(f);

eName.Text = ""; // clear for next entry

eDescription.Text = "";

};

layout.Children.Add(eName);

layout.Children.Add(eDescription);

layout.Children.Add(btnNew);

Content = layout;

# Table View Example

We’ll modify our ListView example to have more functionality in the Fruit Detail Page constructor. The detail page will present a form that will allow modification of a Fruit.

Title = fruit.Name;

StackLayout layout = new StackLayout { HorizontalOptions = LayoutOptions.Center };

TableView table = new TableView { Intent = TableIntent.Form };

EntryCell eName = new EntryCell { Label = "Name", Text = fruit.Name };

EntryCell eDescription = new EntryCell { Label = "Description", Text = fruit.Description };

TableSection section = new TableSection(fruit.Name)

{

eName, eDescription

};

table.Root = new TableRoot { section};

Button btnSave = new Button { Text = "Save Changes" };

btnSave.Clicked += (sender, e) =>

{

fruit.Name = eName.Text;

fruit.Description = eDescription.Text;

Navigation.PopAsync();

};

Button btnCancel = new Button { Text = "Cancel" };

btnCancel.Clicked += (sender, e) => { Navigation.PopAsync(); };

layout.Children.Add(table);

layout.Children.Add(btnSave);

layout.Children.Add(btnCancel);

Content = layout;

The above code will allow edits. The only problem is the change to the fruit will not update the list, but going into the same fruit again will show the changes were saved.

If that is desired you can implement INotifyPropertyChanged for the Fruit class and define a property and a method:

public event PropertyChangedEventHandler PropertyChanged;

private void OnPropertyChanged([CallerMemberName] string property = "")

{

if(PropertyChanged != null)

{

PropertyChanged(this, new PropertyChangedEventArgs(property));

}

}

In each setter:

set

{

if (value != this.fName)

{

this.fName = value;

OnPropertyChanged();

}

}

Note that calling of the property changed event is normally handled automatically, but because we’re basically handling the binding ourselves we need to call it so that the Collection knows there was a change. The collection automatically listens for changes to the items, if the items implement the notify property changed interface.

# ListView Pull To Refresh

A built in behavior in the Xamarin Forms Listview is the pull to refresh. We can respond to the Refreshing event and set the ItemSource again. We’ll create a listview that shows Time information and a pull to refresh will have it update. This is another approach to ensure the List View is in sync, though it is relying on the user to perform the pull and refresh gesture.

Add a second ListView to the layout on the Fruit List Page, enable pull to refresh:

var listView2 = new ListView

{

ItemsSource = GetTime(),

IsPullToRefreshEnabled = true

};

Add a method (just a nonsense one) that will return a list that keeps changing (use system time)

private List<string> GetTime()

{

List<string> result = new List<string>();

result.Add(DateTime.Now.ToString());

result.Add(DateTime.Now.ToString());

result.Add(DateTime.Now.ToString());

result.Add(DateTime.Now.ToString());

return result;

}

Add the Refreshing event handler and add the list view to the layout:

listView2.Refreshing += (sender, e) => {

listView2.ItemsSource = GetTime();

listView2.EndRefresh();

};

layout.Children.Add(listView2);

Content = layout;