Day 19: ListView and Adapters, Part 2

This is part 2 of 2 part series exploring ListView and Adapters in Xamarin.Android in our 31 Days of Xamarin.Android Blog series.

In today’s post we will explore using BaseAdapter and also a custom layout for the ListView’s Item Row.

Let’s dive right into the code and look at what ListView’s AXML and Custom Item Row’s AXML would look like –

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  xmlns:tools="http://schemas.android.com/tools"  android:orientation="vertical"  android:layout\_width="fill\_parent"  android:layout\_height="fill\_parent"  tools:actionBarNavMode="tabs">  <ListView  android:id="@+id/moviesListView"  android:layout\_width="match\_parent"  android:layout\_height="wrap\_content" />  </LinearLayout> |

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  android:orientation="vertical"  android:layout\_width="fill\_parent"  android:layout\_height="fill\_parent"  android:padding="8dp">  <TextView  android:id="@+id/titleTextView"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:textSize="20dp"  android:textStyle="bold"  android:paddingLeft="5dp" />  <TextView  android:id="@+id/directedByTextView"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:textColor="#00A14B"  android:paddingLeft="5dp" />  <TextView  android:id="@+id/releasedDateTextView"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:textColor="#7F3F97"  android:paddingLeft="5dp" />  </LinearLayout> |

Gist file link: <https://gist.github.com/vkoppaka/1744b7ec5fec43b8ee55>

Lets take a look at how the Main Activity that shows the ListView is –

|  |
| --- |
| using Android.App;  using Android.OS;  using Android.Widget;  namespace AdapterDemo2  {  [Activity(Label = "AdapterDemo2",  MainLauncher = true,  Theme = "@android:style/Theme.Holo.Light",  Icon = "@drawable/icon")]  public class MainActivity : Activity  {  protected override void OnCreate(Bundle bundle)  {  base.OnCreate(bundle);  // Set our view from the "main" layout resource  SetContentView(Resource.Layout.Main);  var moviesListView = FindViewById<ListView>(Resource.Id.moviesListView);  moviesListView.ItemClick += moviesListView\_ItemClick;  var moviesAdapter = new MovieAdapter(this, MoviesRepository.Movies);  moviesListView.Adapter = moviesAdapter;  }  void moviesListView\_ItemClick(object sender, AdapterView.ItemClickEventArgs e)  {  Toast.MakeText(this, MoviesRepository.Movies[e.Position].ToString(), ToastLength.Long).Show();  }  }  } |

Gist file link: <https://gist.github.com/vkoppaka/933084220257e9bfb267>

The main difference in this Activity vs what we covered yesterday was that we are no longer using ArrayAdapter but we now have our own custom MovieAdapter class, and the ListView’s Adapter is set to this Adapter.

So what exactly is in the MovieAdapter Class? Let’s explore –

|  |
| --- |
| using System.Collections.Generic;  using Android.App;  using Android.Views;  using Android.Widget;  namespace AdapterDemo2  {  public class MovieAdapter : BaseAdapter<Movie>  {  private readonly Activity context;  private readonly List<Movie> movies;  public MovieAdapter(Activity context, List<Movie> movies)  {  this.context = context;  this.movies = movies;  }  public override Movie this[int position]  {  get  {  return movies[position];  }  }  public override int Count  {  get  {  return movies.Count;  }  }  public override long GetItemId(int position)  {  return position;  }  public override View GetView(int position, View convertView, ViewGroup parent)  {  var view = convertView;  if (view == null)  {  view = context.LayoutInflater.Inflate(Resource.Layout.MovieRow, parent, false);  }  var titleTextView = view.FindViewById<TextView>(Resource.Id.titleTextView);  var directedByTextView = view.FindViewById<TextView>(Resource.Id.directedByTextView);  var releasedDateTextView = view.FindViewById<TextView>(Resource.Id.releasedDateTextView);  titleTextView.Text = movies[position].Title;  directedByTextView.Text = "Directed by: " + movies[position].Director;  releasedDateTextView.Text = "Released on: " + movies[position].ReleaseDate.ToShortDateString();  return view;  }  }  } |

Gist file link: <https://gist.github.com/vkoppaka/ad40e7631abf6bb33bb9>

The MovieAdapter class which inherits from BaseAdapter mainly defines 4 methods that BaseAdapter abstract class mandates. They are –

1. GetItemId(int position)
2. GetView(int position, View convertView, ViewGroup parent)
3. Count
4. this[int position]

## GetItemId

The GetItemId gives you an option to let ListView know what the id of the current item is looked up by position.

## Count

Count property is pretty straight forward, it tells us how many items the ListView is currently showing.

## this[int position]

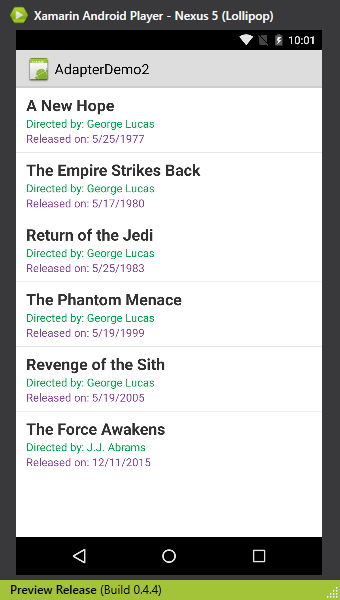
This is .NET’s array indexer overload method which exposes the object at a given position

## GetView(int position, View convertView, ViewGroup parent)

The GetView method is THE most important method in an Adapter. The implementation of GetView starts with getting the View that it will be working with. All View’s in Android are inflated using LayoutInflater and GetView method is no different. Using the LayoutInflater we will identity which Layout, MovieRow in our context, to inflate.

Once we have the View that we will be working with inflated we can then use the regular FindViewById methods that we are already familiar with and set text, and possibly any other properties on the Views.

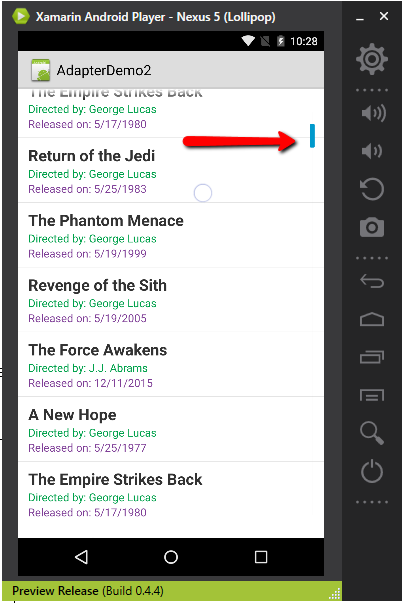
Let’s run the application now and see how it looks –



Ooh, pretty. This is just the beginning of what a custom ListView and Adapter can do and the opportunities on what you can do with it are endless.

# Fast Scrolling

Fast Scrolling in a list view is extremely useful if there are lots of data that your ListView is showing. With fast scrolling you drag the scroll bar even more fast to get through lots of data.



To enable Fast Scroll, all you have to do is

|  |
| --- |
| moviesListView.FastScrollEnabled = true; |

# Section Indexing

Now having fast scrolling is great, but what would make scrolling through lots of rows of data easy is something called Section Indexing. To enable Section Indexing, you will be inheriting your Activity from “**ISectionIndexer**”

Lets see how the Adapter looks like after implementing methods mandated by ISectionIndexer –

|  |
| --- |
| using System.Collections.Generic;  using Android.App;  using Android.Views;  using Android.Widget;  namespace AdapterDemo2  {  public class MovieAdapter : BaseAdapter<Movie>, ISectionIndexer  {  private readonly Activity context;  private readonly List<Movie> movies;  public MovieAdapter(Activity context, List<Movie> movies)  {  this.context = context;  this.movies = movies;  }  public override Movie this[int position]  {  get  {  return movies[position];  }  }  public override int Count  {  get  {  return movies.Count;  }  }  public override long GetItemId(int position)  {  return position;  }  public override View GetView(int position, View convertView, ViewGroup parent)  {  var view = convertView;  if (view == null)  {  view = context.LayoutInflater.Inflate(Resource.Layout.MovieRow, parent, false);  }  var titleTextView = view.FindViewById<TextView>(Resource.Id.titleTextView);  var directedByTextView = view.FindViewById<TextView>(Resource.Id.directedByTextView);  var releasedDateTextView = view.FindViewById<TextView>(Resource.Id.releasedDateTextView);  titleTextView.Text = movies[position].Title;  directedByTextView.Text = "Directed by: " + movies[position].Director;  releasedDateTextView.Text = "Released on: " + movies[position].ReleaseDate.ToShortDateString();  return view;  }  Java.Lang.Object[] sectionHeaders = SectionIndexerBuilder.BuildSectionHeaders(MoviesRepository.Movies);  Dictionary<int, int> positionForSectionMap = SectionIndexerBuilder.BuildPositionForSectionMap(MoviesRepository.Movies);  Dictionary<int, int> sectionForPositionMap = SectionIndexerBuilder.BuildSectionForPositionMap(MoviesRepository.Movies);  public Java.Lang.Object[] GetSections()  {  return sectionHeaders;  }  public int GetPositionForSection(int section)  {  return positionForSectionMap[section];  }  public int GetSectionForPosition(int position)  {  return sectionForPositionMap[position];  }  }  } |

Gist file link: <https://gist.github.com/vkoppaka/bd7aedbbd0153fa90d27>

The Section Indexer methods that the activity should be implementing are –

1. GetSections()
2. GetPositionForSection()
3. GetSectionForPosition()

## GetSections()

Let’s Android know of all the Sections the Listview should be showing.

## GetPositionForSection()

Gets the integer position for a given section.

## GetSectionForPosition()

Gets the section for a given position.

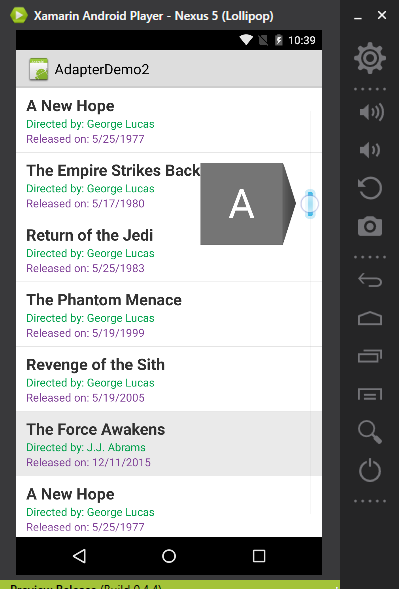
Now all these three methods make use of SectionIndexBuilder, a class I borrowed from Xamarin’s tutorials. Let’s see what SectionIndexBuilder does –

|  |
| --- |
| using System.Collections.Generic;  namespace AdapterDemo2  {  public static class SectionIndexerBuilder  {  // builds an array of unique section headers, data must be sorted by name  public static Java.Lang.Object[] BuildSectionHeaders(List<Movie> data)  {  var results = new List<string>();  var used = new SortedSet<string>();  foreach (var item in data)  {  var letter = item.Title[0].ToString();  if (!used.Contains(letter))  results.Add(letter);  used.Add(letter);  }  var jobjects = new Java.Lang.Object[results.Count];  for (int i = 0; i < results.Count; i++)  {  jobjects[i] = results[i];  }  return jobjects;  }  // builds a map to answer: position --> section, data must be sorted by name  public static Dictionary<int, int> BuildSectionForPositionMap(List<Movie> movies)  {  var results = new Dictionary<int, int>();  var used = new SortedSet<string>();  int section = -1;  for (int i = 0; i < movies.Count; i++)  {  var letter = movies[i].Title[0].ToString();  if (!used.Contains(letter))  {  section++;  used.Add(letter);  }  results.Add(i, section);  }  return results;  }  // builds a map to answer: section --> position, data must be sorted by name  public static Dictionary<int, int> BuildPositionForSectionMap(List<Movie> movies)  {  var results = new Dictionary<int, int>();  var used = new SortedSet<string>();  int section = -1;  for (int i = 0; i < movies.Count; i++)  {  var letter = movies[i].Title[0].ToString();  if (!used.Contains(letter))  {  section++;  used.Add(letter);  results.Add(section, i);  }  }  return results;  }  }  } |

Gist file link: <https://gist.github.com/vkoppaka/3060894c85eeeac06fa9>

All that SectionIndexBuilder does is some smart things with the List data to find out what sections and positions for a given list of data is.

Now, if you were to run the application, you should start seeing “sections” which makes it even more easier for your users to scroll through large lists of data.



That is a wrap with ListView and Adapters in Xamarin.Android, I’ll see you all tomorrow.

Venkata