Cross platform mobile apps using .NET

Table of content

1. Introduction

1.1. About this tutorial

With mobile taking off in a big way it is a fun and exciting time to be a software developer. The landscape is rapidly changing, with a wide variety of options for platforms and programming languages. Businesses are faced with tough decisions on how to provide a best possible user experience, yet keeping maintenance cost down across the different smart phone platforms.

Some are compromising user experience and betting on web based interfaces, while others require the high fidelity user experience or device integration traditionally only found in native apps.

In this workshop we will explore how we can write fully native applications taking full advantage of the platform, yet achieving a high level of code reuse across Windows Phone 7, Android and iOS.

Some of the topics covered in the workshop:

- Separated Presentation patterns for maximum code reuse across all platforms
- How to structure your projects and build for the different platforms
- How to access device specific functionality like GPS and Camera in a cross platform way
- An end-to-end example of cross platform mobile apps in practice (Flights Norway – monitor arrivals and departures from any airport in Norway)

The tutorial we will start with some basic examples of code-reuse across the three platforms to illustrate how to build class libraries and link source files. Later we will cover more advanced topics, like how to re-use as much of the UI-layer as possible, how to patch differences in the .NET framework available on each platform, how to abstract platform specific concepts like thread marshaling, and finally how to consume device specific features such as GPS.

1.2. Installation instructions

In order to complete this workshop you must install the necessary software. If you use a Mac I would recommend installing MonoTouch and Mono for Android. If you were using a PC you will have to install the Windows Phone 7 tools and Mono for Android. Detailed installation instructions can be found on the individual landing pages for each toolkit.

- MonoTouch installation instructions: http://monotouch.net/Documentation/Installation
- Mono for Android installation instructions: http://mono-android.net/Installation
- Windows Phone 7 installation instructions: http://create.msdn.com/en-us/home/getting_started

Once you have the necessary software installed it's recommended that you verify your development environment by doing a simple "Hello World" application. Make sure you can build and run a "default" MonoTouch/Mono for Android app, as well as a Windows Phone 7 app.

1.3. Example code and github

The code examples for this workshop are available at http://github.com/follesoe/FlightsNorway. It is possible to download the code directly, however you might want to install Git in order to clone the repository, as well as access pre-defined branches with "clean" starting points for each step of the workshop. Using Git will also make it easier to move code between operating systems if you are developing for iOS using MonoTouch.

Github got an excellent guide for how to configure Git for Mac (http://help.github.com/mac-set-up-git) and Windows (http://help.github.com/win-set-up-git).

2. Using Class Libraries to reuse code

Class libraries exist to separate code into reusable components. Most commonly the class libraries are used across the same type of project, such as a web application or a desktop application. In this case we want to share class libraries across multiple devices, and unfortunately different class library projects are needed for MonoTouch, Mono for Android and Windows Phone 7.

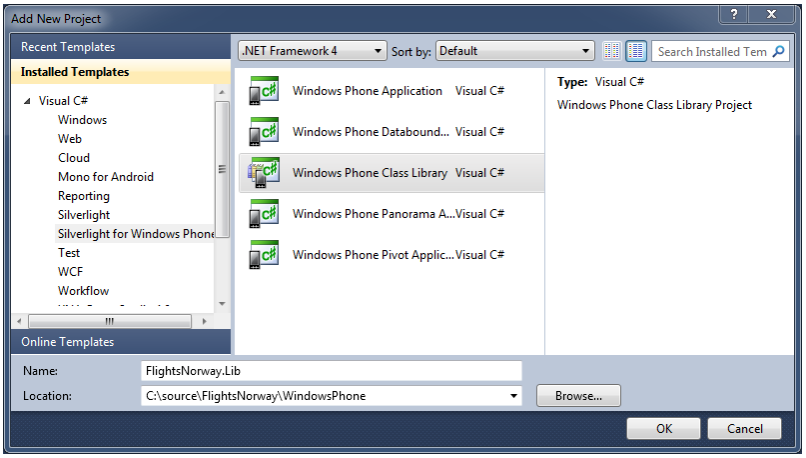
The first part of this tutorial is to get familiarized with developing for multiple platforms. As an example we will develop a simple application reading a text file containing information about airports in Norway. We read the files into a list of Airport-objects, which we will display on the screen. The class representing an Airport, as well as the code to read the data file will be reused.

2.1. Creating solution folders and Windows Phone 7 projects

1. Start by creating a new blank Visual Studio 2010 solution and name it "FlightsNorway".

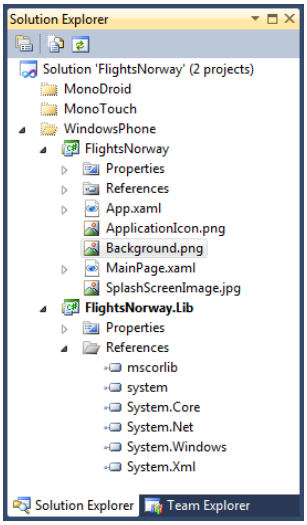
2. Add two solution folders to the solution, "WindowsPhone" and "MonoDroid".

3. Add a new project to the "WindowsPhone" solution folder. Select the "Windows Phone Class Library" project type, and name the project "FlightsNorway.Lib". Make sure the location matches your solution folder name. I.e. c:\source\FlightsNorway\WindowsPhone.



4. Add a new project to the "WindowsPhone" solution folder. This time you should select the "Windows Phone Application" project type, and simply name the project "FlightsNorway". Make sure the location of the project matches your solution folder structure.

5. Add a project reference form the "FlightsNorway" app to the "FlightsNorway.Lib" class library. Your solution structure should look something like this:



2.2. Reading XML file with Norwegian airports

Now that you got the basic project structure setup we can start adding functionality. The first thing we are going to add is a XML file containing a list of all Norwegian airports, with name, code as well as location coordinates. The data file can be downloaded from: https://github.com/follesoe/FlightsNorway/raw/read-airports-from-file/WindowsPhone/FlightsNorway/Content/Airports.xml.

1. Add a new folder to the "FlightsNorway" project called "Content". Add the downloaded Airports.xml file. Be sure to mark it with build type "Resource" to be built as an embedded resource.

2. The next thing we need is a class representing an Airport. Start by adding a new folder to the "FlightsNorway.Lib" project called "Model". Add a new class called "Airport" to the "Model" folder.

3. We also need a class representing a geo location. Add a new class called "Location" to the "Model" folder.

4. Implement the Airport and Location classes. The Airport should hold a Code and a Name, while the Location should hold two numbers (double) called Latitude and Longitude. Se code listing 1 and 2 for implementation details.

Code listing 1 – Airport.cs

```
public class Airport
{
    public string Code { get; set; }
    public string Name { get; set; }
    public Location Location { get; set; }

    public override string ToString()
    {
        return string.Format("{0} - {1}", Code, Name);
    }
}
```

Code listing 2 – Location.cs

```
public class Location
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}
```

With classes to represent Airport and Location we can now parse the content of the Airports.xml file. To do this we are going to use XLinq.

1. Start by adding a reference to "System.Xml.Linq" from the "FlightsNorway.Lib" project.

2. Add a new folder to the "FlightsNorway.Lib" project called "DataServices".

3. Add a new class to the "DataServices" folder called "AirportNamesService". This class will be responsible for parsing the XML file, as well as getting airport names from a REST service later in the tutorial.

4. The "AirportNamesService" should have a method called "GetNorwegianAirports", that returns a list of Airport-objects. The method should accept a stream as the parameter (a stream holding the content of the XML file). Se code listing 3 for implementation details.

Code listing 3 – AirportNamesService.cs

```
using System;
using System.Linq;
using System.Xml.Linq;
using System.Collections.Generic;

using FlightsNorway.Lib.Model;

namespace FlightsNorway.Lib.DataServices
{
    public class AirportNamesService
    {
        public List<Airport> GetNorwegianAirports(Stream stream)
        {
            var xml = XDocument.Load(stream);

            return (from a in xml.Root.Descendants("Airport")
                    select new Airport
                    {
                        Code = a.Attribute("Code").Value,
                        Name = a.Attribute("Name").Value,
                        Location = new Location
                        {
                            Latitude = Convert.ToDouble(a.Attribute("Lat").Value),
                            Longitude = Convert.ToDouble(a.Attribute("Lon").Value)
                        }
                    }).ToList();
        }
    }
}
```

2.3. Displaying airports on Windows Phone 7

Now that we got code to parse the Airports.xml file we can start adding some simple UI to the app. To begin with we will simply list all the Norwegian airports. Open the "MainPage.xaml" file from the "FlightsNorway" project. This fill contains the main page of the application. Inside the file you will find a Grid called "ContentPanel". Add a "ListBox" inside the "ContentPanel" and name the ListBox "_airports". This will give us programmatic access to the ListBox in the code-behind file. Se code listing 4 for an example of what the XAML should look like.

Code listing 4 – MainPage.xaml

```
<!--ContentPanel - place additional content here-->
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <ListBox x:Name="_airports" />
</Grid>
```

Next we need to use the "AirportNamesService" to get a list of all Norwegian airports and set these as the "ItemsSource" on the "ListBox".

1. Open the "MainPage.xaml.cs" file and add an event handler for the "Loaded"-event in the constructor.

2. Inside the event handler method create a new instance of the "AirportsNameService".

3. Create a Uri object that points to the Airports.xml file.

4. Calling the "GetNorwegianAirports" method and set the return value of the method as the "ItemsSource" on the "ListBox". See code listing 5 for implementation details.

Code listing 5 – MainPage.xaml.cs

```
// Constructor
public MainPage()
{
    InitializeComponent();
    Loaded += MainPage_Loaded;
}

private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    var service = new AirportNamesService();

    var fileUri = new Uri("FlightsNorway;component/Content/Airports.xml",
                          UriKind.Relative);

    using (var stream = Application.GetResourceStream(fileUri).Stream)
    {
        _airports.ItemsSource = service.GetNorwegianAirports(stream);
    }
}
```

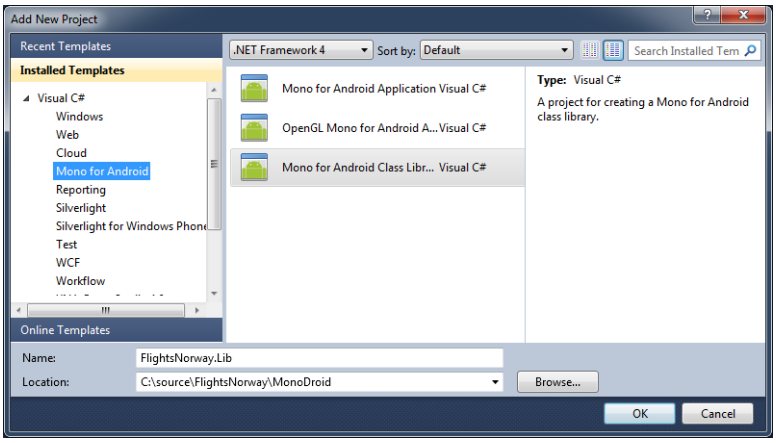
When you run the app in the Windows Phone 7 Emulator you will see a UI that looks something like this:

As you probably will notice some of the airport names are messed up. For example Brønnøysund is listed as "Br%C3%B8nn%C4%B8ysund". The reason for this is that all airport names in the Airports.xml file are url encoded (for no other reason than to demonstrate a point later in this tutorial).

2.4. Creating Mono for Android projects and linking files

Now that we got our basic app running on Windows Phone 7 we can start implementing it for Android. The first thing we need to do is to create a new class library for the "Airport", "Location" and "AirportNamesService" classes. We also need a project for the app itself.

1. Add a new project to the "MonoDroid" solution folder. Select the "Mono for Android Class Library" project type, and name the project "FlightsNorway.Lib". Make sure the location matches your solution folder name. I.e. c:\source\FlightsNorway\MonoDroid.

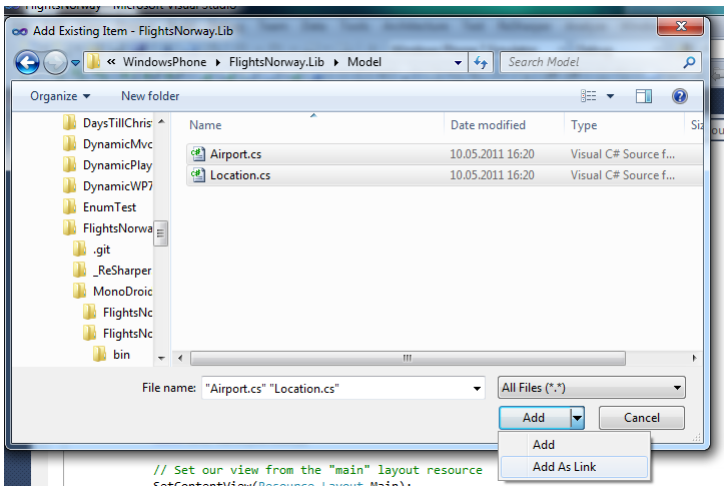


2. Add a new project to the "MonoDroid" solution folder. This time you should select the "Mono for Android Application" project type, and simply name the project "FlightsNorwayDroid". Make sure the location of the project matches your solution folder structure.

3. Add a project reference form the "FlightsNorwayDroid" app to the "FlightsNorway.Lib" class library.

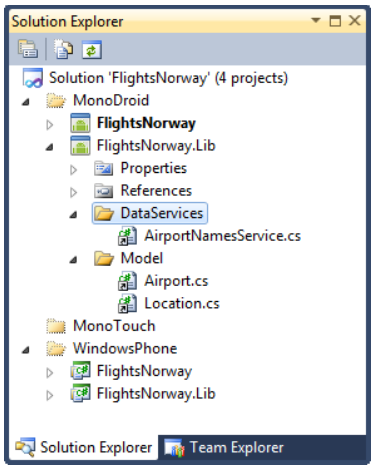
4. Add two new folders to the "FlightsNorway.Lib" project, "Model" and "DataServices". We need these two folders to match the structure of the Windows Phone 7 class library.

5. Right click the "Model" folder and select "Add existing item". This will bring up a file dialog. Navigate to the location of the Windows Phone 7 class library, and find the "Airport" and "Location" files in the Model directory. Do not click "Add" like you normally would. Instead click the small arrow on the button and select "Add As Link".



6. Do the same for the "AirportsNameService". You also need to add a reference to "System.Xml.Linq".

When adding a file as a link you does not make a new copy of the file, instead you are simply referencing the exact same file as the Windows Phone 7 class library. If you make a change to any of the files, the change will be reflected in both class libraries when you build the project. After completing these steps your solution structure should look like this:



Notice how the little arrow on the icons for "AirportNamesService", "Airport" and "Location" indicate that the files linked.

2.5. Displaying airports on Android

Now that we got our class library set up for Android we can start implementing the UI. The structuring of Android apps is quite different than Windows Phone 7. The first thing we need to do is include the Airports.xml file. Add it as a link in the Assets folder. Mono for Android will automatically make sure the file has the right build type.

Next we need to change "Activity1.cs", the entry point of the application. The class should derive from "ListActivity" (as it is going to display a list of Airports). We also need to change the OnCreate-method to set a "ListAdapter", which is responsible for populating the list. Android has a built in "ArrayAdapter" we can use to display our list of Airports. Code listing 6 includes the implementation details.

Code listing 6 – Activity1.cs

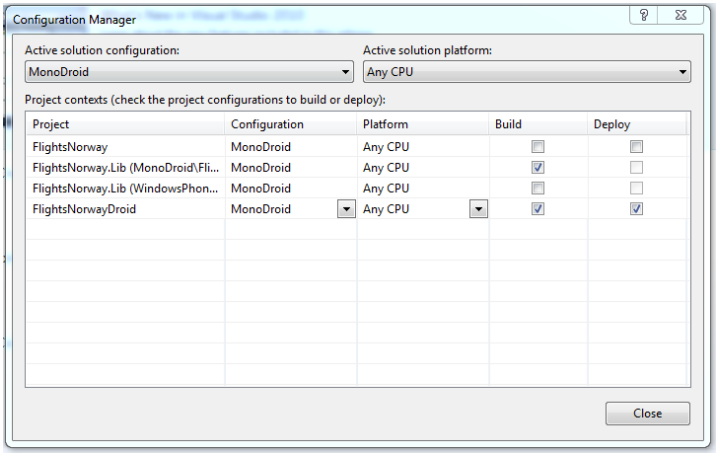
```
[Activity(Label = "FlightsNorway", MainLauncher = true, Icon = "@drawable/icon")]
public class Activity1 : ListActivity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);

        var airportsService = new AirportNamesService();
        using(var stream = Assets.Open("Airports.xml"))
        {
            var airports = airportsService.GetNorwegianAirports(stream);
            ListAdapter = new ArrayAdapter<Airport>(this,
                                Android.Resource.Layout.SimpleListItem1, airports);
        }
    }
}
```

2.6. Using Build Configuration to separate Android and Windows Phone 7

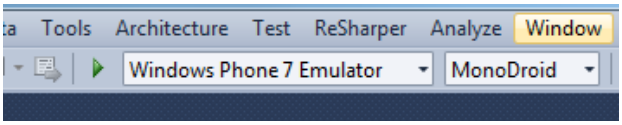
In order to work with both Windows Phone 7 and Mono for Android in the same solution you need to create some custom build configurations.

1. Open the Build Configuration Manager by clicking "Build – Configuration Manager"

2. The active configuration is set to Debug – click the dropdown and select "New…". In the dialog that pops up, name the new build configuration "MonoDroid", and choose to copy settings from the Debug build. Check the "Deploy" checkbox for the "FlightsNorwayDroid" project and the "FlightsNorway.Lib" Mono for Android project. Uncheck the Deploy checkbox for the Windows Phone 7 project.



3. Select the MonoDroid build configuration and run the app.

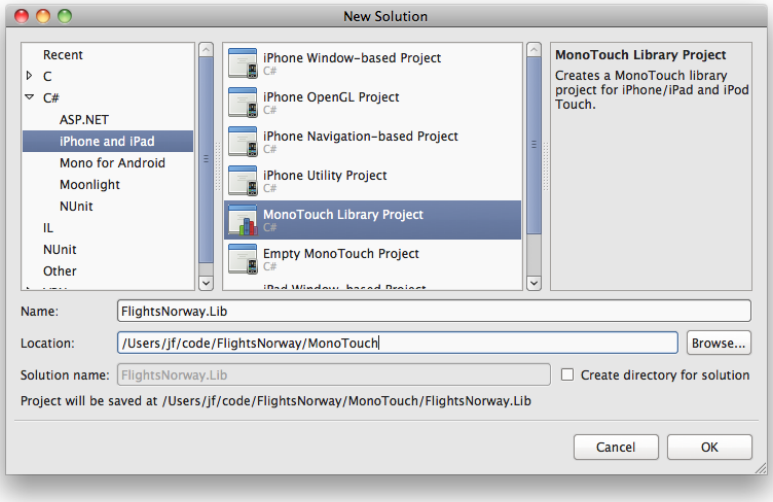


When you run the Android app it should look something like this:

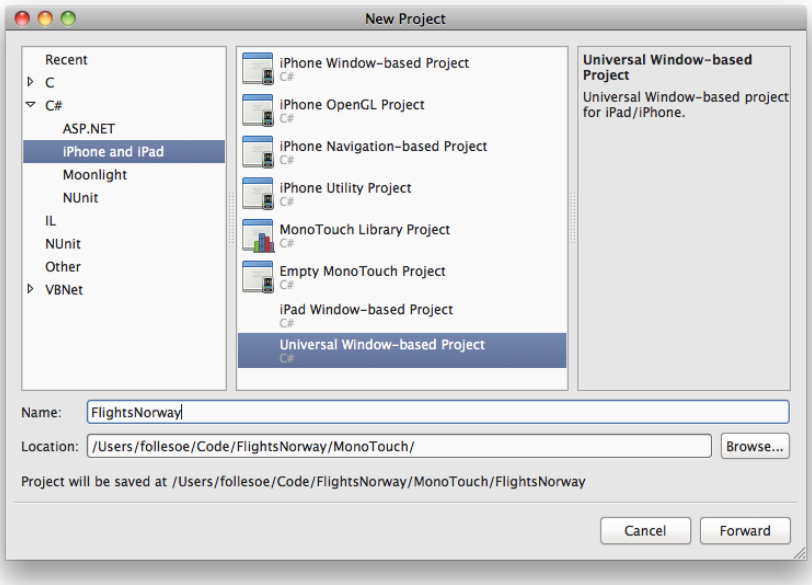
2.7. Creating MonoTouch projects and linking files

Now that we got our basic app running on Windows Phone 7 and Android we can start implementing it for iOS. The first thing we need to do is to create a new class library for the "Airport", "Location" and "AirportNamesService" classes. We also need to create the project for the app itself.

1. Start by creating a new project. Select the "MonoTouch Library Project" project type, and make sure the name, location and solution name is correct.

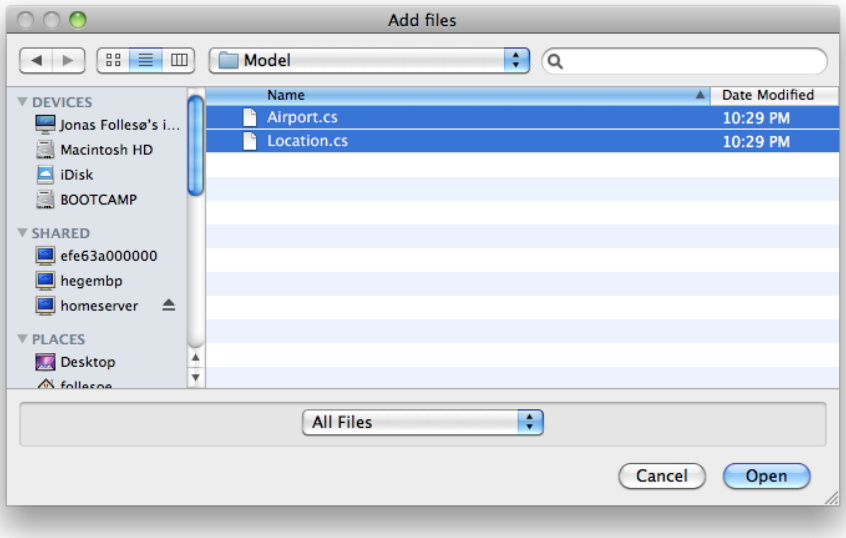


2. Next we need to add the project for the actual app. Add a new project of type "Universal Window-based Project". This will enable us to create a single iOS application targeting iPhone, iPod Touch and iPad.



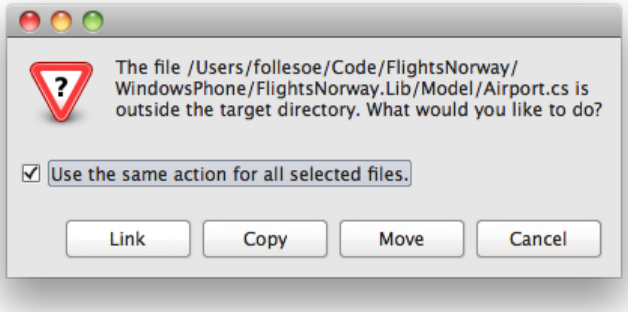
3. Add a reference from "FlightsNorway" to the "FlightsNorway.Lib" class library.

4. Add the "Model" and "DataServices" folders to the "FlightsNorway.Lib" project. Right click the "Model" folder and select "Add files...". This will bring up the following dialog:



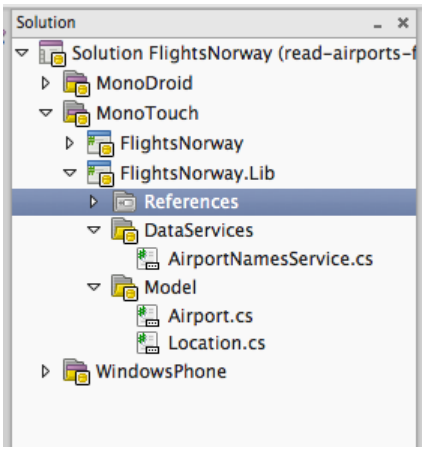
Navigate to the Model folder underneath the Windows Phone 7 project location. Select both "Airport.cs" and "Location.cs" and hit open.

5. Linking files in MonoTouch is slightly different than in Visual Studio 2010. After clicking Open you will be presented with the following dialog:



Check the "Use the same action for all selected files" checkbox, and click Link. This will link the files. Repeat this process for the "AirportNamesService.cs" file.

6. After linking the necessary files the project structure should look something like this:



The tiny "..." box in the bottom right corner of the file icon indicates that the files are linked. The final step is to set the build action for each of the linked ".cs" files to "Compile". This can be a bit cumbersome, as you have to do it for each file...

2.8. Displaying Airports on iOS

With our MonoTouch library set up we can start implementing the UI for displaying airports on iOS. In MonoTouch the main class is called "AppDelegateIPhone" and "AppDelegateIPad" depending on which platform the app is executed. The "Universal Windows-based Project" template will make sure the correct class is invoked when the app starts. To begin with we will focus on iPhone, but the code we will implement will run just as fine on the iPad.

One of the core UI components on the iOS platform is the "TableView", and we will be using it extensively throughout this application.

1. Add a new "Content"-folder to the "FlightsNorway" project. Link in the "Airports.xml" file from the Windows Phone 7 project. Set the build action to "Content".

2. Add a new folder to the "FlightsNorway" project. Name the folder "Tables". This folder will hold all your "TableViewController" classes.

3. Add a new class to the folder called "AirportsTableViewController". Make the class derive from "UITableViewController".

4. Define a new private inner class inside it called "AirportsDataSource". Make the class derive from "UITableViewDataSource". The "DataSource" calss is responsible for providing the data the table is supposed to display. By now your class should look something like this:

Code listing 7 – AirportsTableViewController.cs definition

```
public class AirportsTableViewController : UITableViewController
{
        public AirportsTableViewController ()
        {
        }

        public override void ViewDidLoad ()
        {

        }

        private class AirportsDataSource : UITableViewDataSource
        {
        }
}
```

5. Next thing you need to do is set the "DataSource" property on the "TableView" property of the "AirportsTableViewController". Override the "ViewDidLoad" method on the view controller, and set the "TableView" property.

Code listing 8 – AirportsTableViewController.cs overriding ViewDidLoad

```
public override void ViewDidLoad ()
{
        base.ViewDidLoad ();

        TableView.DataSource = new AirportsDataSource();
}
```

6. The "UITableViewDataSource" class has two abstracts methods you need to implement. The first method is "RowsInSection", which should return the numbers of rows to draw in the table. Implement this method by returning the numbers of airports. You can load the list of airports in the constructor of the "AirportsDataSource" class.

Code listing 9 – AirportsDataSource constructor and RowsInSection

```
private List<Airport> _airports;
private NSString CellID = new NSString("Airports");

public AirportsDataSource()
{
        var service = new AirportNamesService();
        using(var stream = new FileStream("Content/Airports.xml", FileMode.Open))
        {
                _airports = service.GetNorwegianAirports(stream);
        }
}

public override int RowsInSection (UITableView tableView, int section)
{
        return _airports.Count;
}
```

7. Next you need to implement the "GetCell" method. This method is called by the "TableView" to render the data for a given row. In this first demo we will simply set the content of the row to the string representation of our "Airport" object. The "DequeueReusableCell" is used to reuse existing "UITableViewCell" objects that are scrolled off screen.

Code listing 10 – AirportsDataSource GetCell

```
public override UITableViewCell GetCell (UITableView tableView, NSIndexPath indexPath)
{
        var cell = tableView.DequeueReusableCell(CellID);
        if (cell == null)
        {
            cell = new UITableViewCell(UITableViewCellStyle.Default, CellID);
        }

        cell.TextLabel.Text = _airports[indexPath.Row].ToString();

        return cell;
}
```

Now that we have completed the "AirportsTableViewController" implementation we need to add it to our main screen to display it to the user. To add the table to the UI you need to edit the "AppDelegateIPhone" class (for now we will leave the iPad version). Inside these classes you will find a "FinishedLaunching" method, which will be where you populate the main window of the app. simply create a new instance of the "AirportsTableViewController" and add it to the window before calling the "MakeKeyAndVisible" method.

Code listing 10 – AppDelegateIPhone.cs

```
public partial class AppDelegateIPhone : UIApplicationDelegate
{
        private AirportsTableViewController _airportsTvc;

        public override bool FinishedLaunching (UIApplication app, NSDictionary options)
        {
                _airportsTvc = new AirportsTableViewController();

                window.AddSubview(_airportsTvc.View);
                window.MakeKeyAndVisible ();

                return true;
        }
}
```

When you run the app it should look something like this:

3. Using preprocessor directives to url decode Airport names

As you can see from the screenshot of the iPhone app the airport names are all messed up. The reason for this is that the airport names are url encoded. This is done on purpose to illustrate an example of how you can use preprocessor directives when trying to decode the airport names.

3.1. Decoding Airport names on Windows Phone 7

The code to url decode airport names and codes is pretty straight forward. The "HttpUtility" class got static methods to decode a string. We simply need to call the "UrlDecode"-method when parsing the xml file. Edit the "AirportNamesService" class and call "UrlDecode" when reading name and code from the XML file.

Code listing 11 – UrlDecode Airport names and codes

```
Code = HttpUtility.UrlDecode(a.Attribute("Code").Value),
Name = HttpUtility.UrlDecode(a.Attribute("Name").Value),
```

If you run the Windows Phone 7 app you will see that the airport names are now displayed correctly.

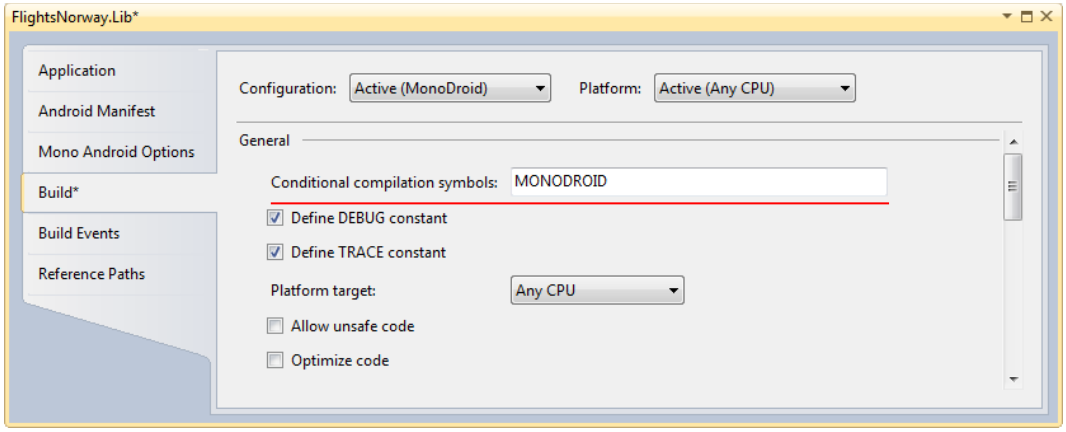
3.2. Decoding Airport names on MonoTouch and Mono for Android

If you try to run the Mono for Android or MonoTouch application you will get a compile error. The reason for this is that "HttpUtility.UrlDecode" is included in the "System.Windows" assembly on Windows Phone 7. This assembly is not present on Mono for Android or MonoTouch. Instead you can add a reference to "System.Web.Services" and add a using directive for "System.Web".

If you try to build the project it should build fine on Mono for Android, but the code no longer works on Windows Phone 7, as we do not have the "System.Web.Services" assembly reference and the project will not be able to resolve the "System.Web" using. To solve this we need to wrap the using in a preprocessor directive so that the "System.Web" using is only compiled on Mono for Android and MonoTouch.

1. Add a reference to "System.Web.Services"

2. Define a new "Conditional compilation symbol" by going into the project properties page under the build tab.



3. Edit the using section of "AirpotNamesService" and wrap the "System.Web" using in a preprocessor directive.

Code listing 12 – Using section with preprocessor directive

```
using System;
using System.IO;
using System.Linq;
using System.Net;
#if MONOTOUCH || MONODROID
using System.Web;
#endif
using System.Xml.Linq;
using System.Collections.Generic;
```

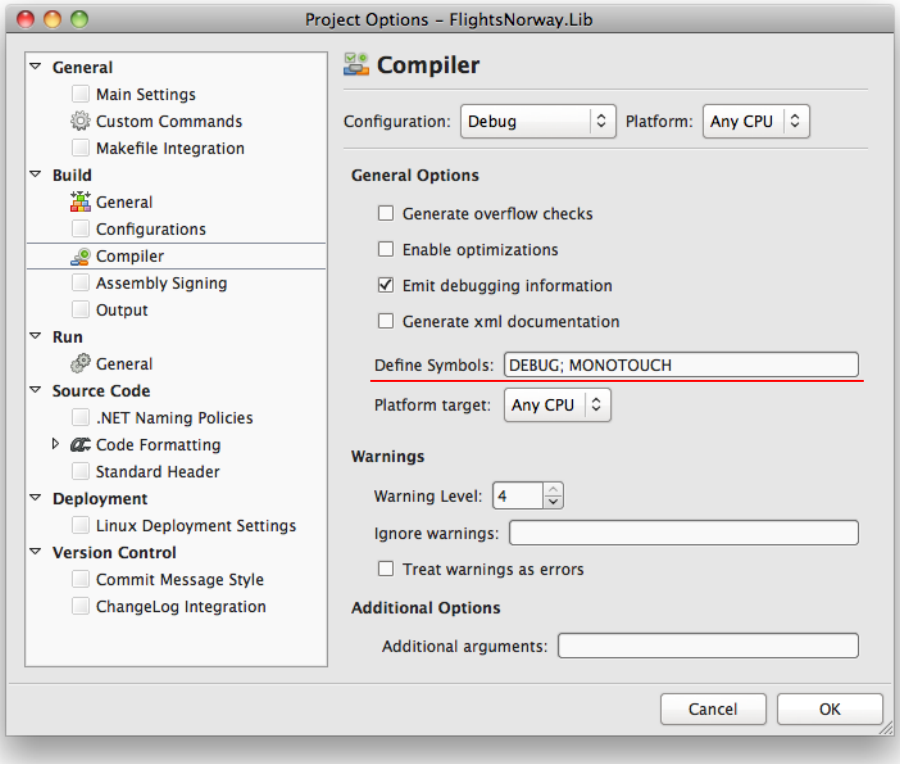
The preprocessor directive will make sure the "System.Web" using is only compiled when the "MONOTOUCH" or "MONODROID" compilation symbols are defined.

3.3. Defining conditional compilation symbols in MonoTouch

To get the url decoding to work on MonoTouch you have to do the same steps as for the Mono for Android version.

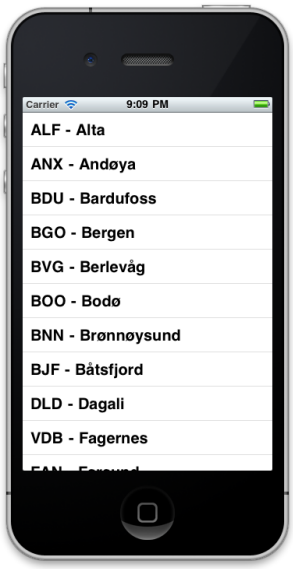
1. Add a reference to "System.Web.Services"

2. Define a new "Conditional compilation symbol" by going into the project properties page under the build tab.



3. Edit the using section of "AirpotNamesService" and wrap the "System.Web" using in a preprocessor directive.

When you run the app you should now se decoded airport names.

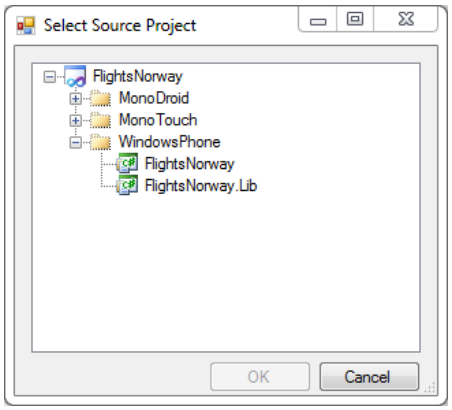


4. Streamlining the cross platform process

4.1. Using the Project Linker extension

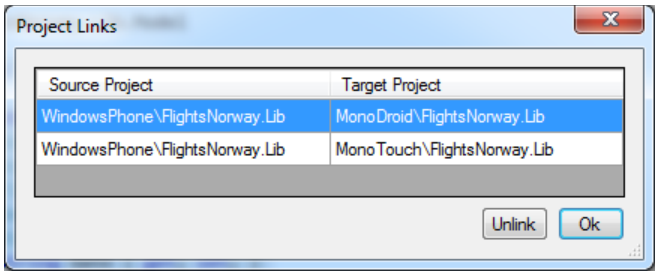
The Project Linker extension is a tool from Microsoft Patterns and Practices. The tool was originally written to make it easier to share code between WPF and Silverlight applications by automatically keeping class libraries in sync. You set up a link between two class libraries, and when you add a new folder or file to the source project, the linked project automatically get a link to the same file. The same tool works great for linking class libraries when doing cross platform mobile development. You can use this tool to link your Windows Phone 7 "FlightsNorway.Lib" project to the Mono for Android and MonoTouch counter parts.

The Project Linker can be installed from the Visual Studio Extension Manager (Tools – Extension Manager). Search for Project Linker. When the extension is installed you can right click a class library and choose "Add project link". This will bring up the following dialog:



Select the "FlightsNorway.Lib" project under the "WindowsPhone" solution folder to link it to your Mono for Android or MonoTouch class library. Whenever you add a new class to the project the file will automatically be linked in to the linked projects.

You can view project inks by clicking "Project – Edit links". This will bring up the following dialog:



As you can see the "WindowsPhone\FlightsNorway.Lib" project links both to the Mono for Android and MonoTouch class libraries.

5. Building the REST client for the Flights service

Now that we've covered the basics of setting up a cross platform mobile project we can start implementing the core features of the app. The app will show arrivals and departures from all Norwegian airports. The data will be retrieved from a REST services exposed by Avinor, the company responsible for running the different airports in Norway. The REST clients will be shared among the Android, iOS and Windows Phone 7 app.

5.1. Adding Model classes

The app needs a way to represent concepts such as Airline, Airport, Flight and Status. Each of the Model classes will be listed as code listings, and needs to be added to the "FlightsNorway.Lib" project for the Windows Phone 7 class library, and linked into the Mono for Android and MonoTouch projects. If you have installed the MonoTouch for Visual Studio extension and the Project Linker this process will be allot smother.

The first model class you need to add is the "CodeName" class. The class is a base class used to represent an item with a code and name. The REST API we are going to call expose things like airlines, airports and status with a code and a name.

Code listing 13 – CodeName.cs

```
public abstract class CodeName
{
    public string Code { get; set; }
    public string Name { get; set; }

    public override string ToString()
    {
        return string.Format("{0} - {1}", Code, Name);
    }
}
```

When parsing Flights from the service we only get the code of the airport, airline or status for the flight. In order to "hook" the Flight up against the correct Airport, Airline or Status object we are going to use a dictionary to look up the objects. The dictionary is going to have som especial behavior, where it will return an "unknown" object if the key does not exist in the dictionary. This way the app won't crash if we are missing a status, airport or airline code.

Code listing 14 – CodeNameDictionary.cs

```
public class CodeNameDictionary<T> :
    Dictionary<string, T> where T : CodeName, new()
{
    public new T this[string code]
    {
        get
        {
            if (ContainsKey(code)) return base[code];
            var item = new T();
            item.Code = code;
            item.Name = "Unknown";
            return item;
        }
        set { base[code] = value; }
    }

    public void AddRange(IEnumerable<T> items)
    {
        foreach (var item in items)
            Add(item);
    }

    public void Add(T item)
    {
        if (ContainsKey(item.Code)) return;

        Add(item.Code, item);
    }
}
```

The dictionary can be used for any class deriving from "CodeName". Next thing you need to add are the classes representing Airline, Airport and Status.

Code listing 15 – Airline.cs

```
public class Airline : CodeName
{
}
```

Code listing 16 – Airport.cs

```
public class Airport : CodeName
{
    public Location Location { get; set; }
}
```

Code listing 17 – Status.cs

```
public class Status : CodeName
{
}
```

Code listing 18 – FlightStatus.cs

```
public class FlightStatus
{
    public Status Status { get; set; }
    public DateTime StatusTime { get; set; }

    public FlightStatus()
    {
        Status = new Status();
    }
}
```

The main model class of the app is the "Flight"-class. It represents a departure or arrival from an airport.

Code listing 19 – Flights.cs

```
public class Flight
{
    public string FlightId { get; set; }
    public Airline Airline { get; set; }
    public Airport Airport { get; set; }
    public FlightStatus FlightStatus { get; set; }
    public DateTime ScheduledTime { get; set; }
    public Direction Direction { get; set; }
    public string Gate { get; set; }
    public string Belt { get; set; }

    public Flight() : this(new Airline(), new Airport())
    {
    }

    public Flight(Airline airline, Airport airport)
    {
        Airline = airline;
        Airport = airport;
        FlightStatus = new FlightStatus();
    }

    public override string ToString()
    {
        return string.Format("{0} from/to {1} at {2}",
            FlightId, Airport.Name, ScheduledTime.ToShortTimeString());
    }
}

public enum Direction
{
    Arrival,
    Depature
}
```

5.2. Base class for REST clients

Now that you have defined the model classes we can start implementing the REST client that is going to download and parse XML with flight information. The API is documented at http://flydata.avinor.no/, and mainly consists of end-points to download reference data about Airports, Status codes and Airlines, as well as the actual Flight service returning up-to-date information about scheduled arrivals and departures at a given airport.

All the REST client classes will share some of the same behavior. You need to make a HTTP GET request against a given resource. If the resource is successfully downloaded the XML needs to be parsed into instances of our model classes. Each REST client will derive from the same base class.

1. Start by adding a new class called "RestService" to the "DataServices" folder. The class will be a generic base class. The class declaration should look something like this:

Code listing 20 – RestService.cs declaration

```
public abstract class RestService<T> where T : class
{
    private string _baseUrl = "http://flydata.avinor.no/";

    public abstract IEnumerable<T> ParseXml(XmlReader reader);
}
```

2. The "RestService" will be used by calling a Get-method, passing in the resource (i.e. Airports.xml) you want to download, as well as a callback delegate. In Windows Phone 7 all networking is asynchronous, so the callback will be invoked when the data is downloaded and parsed. This is a common pattern when doing asynchronous programming in C#. The Get-method should look like this:

Code listing 21 – Get method

```
protected void Get(string resource, ResultCallback<IEnumerable<T>> callback)
{
    var webRequest = (HttpWebRequest)WebRequest.Create(_baseUrl + resource);

    webRequest.BeginGetResponse(responseResult =>
    {
        try
        {
            var response = webRequest.EndGetResponse(responseResult);
            if (response != null)
            {
                var result = ParseResult(response);
                response.Close();
                callback(new Result<IEnumerable<T>>(result));
            }
        }
        catch (Exception ex)
        {
            callback(new Result<IEnumerable<T>>(ex));
        }

    }, webRequest);
}
```

3. The "Get"-method calls a "ParseResult"-method if the web request was successful. The method will read the XML from the response in the correct encoding, before passing the XML on to the abstract method that will turn the XML into instances of T.

Code listing 21 – ParseResult method

```
private IEnumerable<T> ParseResult(WebResponse response)
{
    var encoding = Encoding.GetEncoding("iso-8859-1");
    using (var sr = new StreamReader(response.GetResponseStream(), encoding))
    using (var xmlReader = XmlReader.Create(sr))
    {
        return ParseXml(xmlReader);
    }
}
```

The callback parameter passed to the "Get"-method of the REST client is of type "ResultCallback<IEnumerable<T>>". The "ResultCallback"-delegate is part of the asynchronous pattern used by the app. It forces the callback to accept a parameter of type "Result<T>", which is a way to wrap the result of the asynchronous web request. The result can either be successful and data will be returned, or it can fail with an exception. Since the request is happening on a separate thread we cannot depend on regular exception handling techniques. Instead the exception needs to be handled, and passed back in the callback inside "Result<T>".

Add a new class to the "DataServices" folder called "Result". The implementation should look like this:

Code listing 22 – Result.cs

```
public delegate void ResultCallback<T>(Result<T> result);

public class Result<T>
{
    public Exception Error { get; private set; }
    public T Value { get; private set;  }

    public Result(T value)
    {
        Value = value;
    }

    public Result(Exception error)
    {
        Error = error;
    }

    public bool HasError()
    {
        return Error != null;
    }
}
```

The generic class can be used to represent the result of any asynchronous operation. The result object either accepts the result (T) or an exception in the constructor, depending on weather the call was successful or not. It also has a method to check if the call was successful or not.

5.3. REST clients for Airport, Airline, Status and Flights

Now that we have the REST client base class in place we can start implementing each of the client classes. Start by adding a class to retrieve a list of all known airlines.

Code listing 23 – AirlineNamesService.cs

```
public class AirlineNamesService : RestService<Airline>
{
    public void GetAirlines(ResultCallback<IEnumerable<Airline>> callback)
    {
        Get("airlineNames.asp", callback);
    }

    public override IEnumerable<Airline> ParseXml(XmlReader reader)
    {
        var xml = XDocument.Load(reader);

        return from airlineNames in xml.Elements("airlineNames")
                from airline in airlineNames.Elements("airlineName")
                select new Airline
                            {
                                Code = airline.Attribute("code").Value,
                                Name = airline.Attribute("name").Value
                            };
    }
}
```

The class has a "GetAirlines"-method, which calls the "Get"-method on the generic base class. It also implements the abstract method "ParseXml" which is responsible to turning the returned XML into a sequence of Airline objects.

We already have an "AirportNamesService"-class that returns a list of Norwegian airports with location information. Next you need to add a new method to this class to retrieve a complete list of all airport names and codes. This will be used to display complete airport information for international flights arriving or departing from Norwegian airports. Start by changing the class declaration to derive from "RestService<Airport>", and implement the following methods:

Code listing 25 – New AirportNamesService methods

```
public void GetAirports(ResultCallback<IEnumerable<Airport>> callback)
{
    Get("airportNames.asp", callback);
}

public override IEnumerable<Airport> ParseXml(XmlReader reader)
{
    var xml = XDocument.Load(reader);

    return from airportNames in xml.Elements("airportNames")
            from airport in airportNames.Elements("airportName")
            select new Airport
                        {
                            Code = airport.Attribute("code").Value,
                            Name = airport.Attribute("name").Value
                        };
}
```

Each flight has a status indication if a flight is delayed, arrived, departed, canceled or similar. The status codes and descriptions can also be downloaded from the REST service. To do this you need to add a "StatusService" class.

Code listing 26 – StatusService.cs

```
public class StatusService : RestService<Status>
{
    public void GetStautses(ResultCallback<IEnumerable<Status>> callback)
    {
        Get("flightStatuses.asp", callback);
    }

    public override IEnumerable<Status> ParseXml(XmlReader reader)
    {
        var xml = XDocument.Load(reader);

        return from statuses in xml.Elements("flightStatuses")
                from status in statuses.Elements("flightStatus")
                select new Status
                {
                    Code = status.Attribute("code").Value,
                    Name = status.Attribute("statusTextNo").Value
                };
    }
}
```

The last REST client we need is the one to get the actual flights from a given airport. This is the most comprehensive of all the clients, as this one needs to first get Airports, Airlines and Status-codes before getting the actual Flights, as those objects are needed as reference data to build a complete Flight object. Start by adding a "FlightsService"-class, with the following declaration and class variables.

Code listing 27 – FlightsService declaration and class variables

```
public class FlightsService : RestService<Flight>
{
    private readonly string _resourceUrl;

    private readonly AirportNamesService _airportService;
    private readonly AirlineNamesService _airlineService;
    private readonly StatusService _statusService;

    public CodeNameDictionary<Airline> Airlines { get; private set; }
    public CodeNameDictionary<Airport> Airports { get; private set; }
    public CodeNameDictionary<Status> Statuses { get; private set; }

}
```

As you can see we are referencing the Airport, Airline and Status service. We also use the "CodeNameDictionary"-class defined previously in the tutorial. The dictionaries will be used to look up Airline, Airport and Status objects when parsing the Flight XML data. Next you need to implement the constructor.

Code listing 28 – FlightsService constructor

```
public FlightsService()
{
    Airlines = new CodeNameDictionary<Airline>();
    Airports = new CodeNameDictionary<Airport>();
    Statuses = new CodeNameDictionary<Status>();

    _airlineService = new AirlineNamesService();
    _airportService = new AirportNamesService();
    _statusService = new StatusService();

    _resourceUrl = "XmlFeed.asp?TimeFrom={0}&TimeTo={1}&airport={2}";
}
```

The "_referenceUrl" will be used a string format when generating the URL to request flights from. It takes from and to time, as well as an airport code.

The main method of the class is the "GetFlightsFrom" method. It's responsible for calling the Airport, Airline and Status service before getting flights from the given Airport.

Code listing 29 – FlightsService GetFlightsFrom

```
public void GetFlightsFrom(ResultCallback<IEnumerable<Flight>> callback, Airport
fromAirport)
{
    if(Airports.Count == 0)
    {
        _airportService.GetAirports(r => {
            if(r.HasError()) callback(new Result<IEnumerable<Flight>>(r.Error));
            Airports.AddRange(r.Value);
            GetFlightsIfAllDone(callback, fromAirport);
        });
    }

    if(Airlines.Count == 0)
    {
        _airlineService.GetAirlines(r =>
        {
            if (r.HasError()) callback(new Result<IEnumerable<Flight>>(r.Error));
            Airlines.AddRange(r.Value);
            GetFlightsIfAllDone(callback, fromAirport);
        });
    }

    if (Statuses.Count == 0)
    {
        _statusService.GetStautses(r =>
        {
            if (r.HasError()) callback(new Result<IEnumerable<Flight>>(r.Error));
            Statuses.AddRange(r.Value);
            GetFlightsIfAllDone(callback, fromAirport);
        });
    }

    GetFlightsIfAllDone(callback, fromAirport);
}
```

The method checks weather Airports, Airlines and Statuses has been loaded (number of items in dictionary are zero). If the reference data dictionaries are empty, the different REST services will be called. Each of the service calls passes in a callback lambda, which will add the returned data to the reference dictionaries, as well as checking if all reference data has been loaded. If so, it will call the flights service. This is done in the "GetFlightsIfAllDone"-method.

Code listing 30 – FlightsService GetFlightsIfAllDone

```
private void GetFlightsIfAllDone(ResultCallback<IEnumerable<Flight>> callback, Airport
fromAirport)
{
    if(Airports.Count > 0 && Airlines.Count > 0 && Statuses.Count > 0)
    {
        var resource = string.Format(_resourceUrl, 1, 12, fromAirport.Code);
        Get(resource, callback);
    }
}
```

The method simply checks that all reference data has been loaded, before generating the request url. The url is passed to the "Get"-method of the REST service base class.

The "FlightsService"-class also has to implement the "ParseXml"-method to turn the XML data containing information of flights into Flight objects.

Code listing 30 – FlightsService ParseXml

```
public override IEnumerable<Flight> ParseXml(XmlReader reader)
{
    var xml = XDocument.Load(reader);

    return from airport in xml.Elements("airport")
           from flights in airport.Elements("flights")
           from flight in flights.Elements("flight")
           select new Flight
           {
               FlightId = flight.ElementValueOrEmpty("flight_id"),
               Airline = Airlines[flight.ElementValueOrEmpty("airline")],
               Airport = Airports[flight.ElementValueOrEmpty("airport")],
               FlightStatus = ReadStatus(flight.Element("status")),
               Gate = flight.ElementValueOrEmpty("gate"),
               Belt = flight.ElementValueOrEmpty("belt"),
               Direction = ConvertToDirection(flight.Element("arr_dep")),
               ScheduledTime = flight.ElementAsDateTime("schedule_time")
           };
}
```

The "ParseXml"-method will read out all the flight nodes and parse them into instances of the "Flight"-class. Each Flight object has an Airport, Airline and Status property which is set by accessing the reference dictionaries. The method uses the "ElementValueOrEmpty"-extension method to make the parsing code simpler. The extension method is used to avoid null reference exceptions if a XML node is missing. The parser also uses the "ConvertToDirection" method to turn a direction string into an enum value, as well as the "ReadStatus"-method to parse the status information.

Code listing 31 – FlightsService ConvertToDirection

```
private static Direction ConvertToDirection(XElement element)
{
    if (element == null) return Direction.Depature;
    return element.Value == "D" ? Direction.Depature : Direction.Arrival;
}
```

Code listing 32 – FlightsService ReadStatus

```
private FlightStatus ReadStatus(XElement element)
{
    if(element == null) return new FlightStatus();

    var flightStatus = new FlightStatus();
    flightStatus.Status = Statuses[element.AttributeValueOrEmpty("code")];
    flightStatus.StatusTime = element.AttributeAsDateTime("time");
    return flightStatus;
}
```

5.4. XML extension methods

When parsing the XML data you need to make sure you don't run into any null reference exceptions for missing nodes or attributes. Combining null checks and LINQ queries to read the data can be a bit cumbersome. One easy way to make this simpler is to add some custom extension methods to the XElement class. These extension methods can do null checks and return a safe default is a given node is missing.

Add a new class to the "DataServices" folder called "XmlExtensions". The implementation should look like this:

Code listing 33 – XmlExtensions.cs

```
public static class XmlExtensions
{
    public static string ElementValueOrEmpty(this XElement element, string elementName)
    {
        var childElement = element.Element(elementName);
        return childElement == null ? string.Empty : childElement.Value;
    }

    public static string AttributeValueOrEmpty(this XElement element, string attributeName)
    {
        var attribute = element.Attribute(attributeName);
        return attribute == null ? string.Empty : attribute.Value;
    }

    public static DateTime ElementAsDateTime(this XElement element, string elementName)
    {
        var childElement = element.Element(elementName);
        return childElement == null ? DateTime.MinValue : Convert.ToDateTime(childElement.Value);
    }

    public static DateTime AttributeAsDateTime(this XElement element, string attributeName)
    {
        var attribute = element.Attribute(attributeName);
        return attribute == null ? DateTime.MinValue : Convert.ToDateTime(attribute.Value);
    }
}
```

As you can see the implementation is fairly straight forward. It checks for null, and if so it returns a safe default. There are also extension methods to turn the content of a XML node or attribute into a DateTime.

5.5. Testing the REST client to get Flights

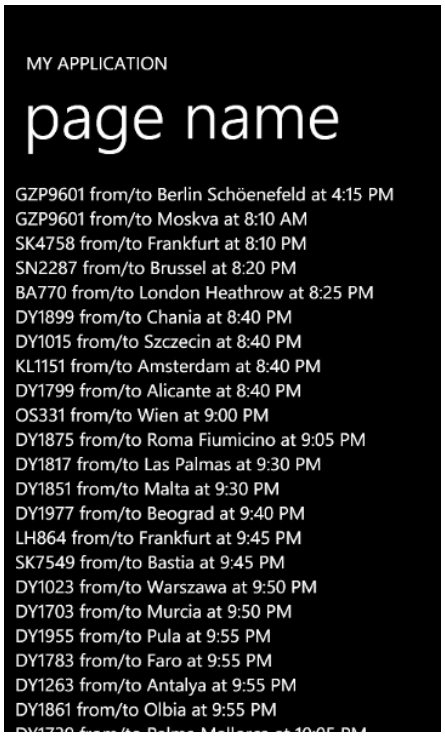
Now that you have implemented the REST clients it's time to test them out. The easiest way to do that is to change the code behind for the "MainPage.xaml" file in the Windows Phone 7 app. Instead of loading airports when the page loads you can try to load flights from a given airport. Change the "MainPage_Loaded"-method to load flights from Oslo airport using the following code:

Code listing 34 – MainPage_Loaded

```
private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    var flightsService = new FlightsService();
    flightsService.GetFlightsFrom(flights =>
                                  {
                                      Deployment.Current.Dispatcher.BeginInvoke(
                                          () => _airports.ItemsSource = flights.Value);
                                  }, new Airport {Code = "OSL"});
}
```

The method creates an instance of the "FlightsService"-class and calls the "GetFlightsFrom"-method. The method accepts a delegate, the callback to call when flights are loaded, as well as the Airport to load flights from. In the callback you need to use the "Dispatcher.BeginInvoke"-method to move from a worker thread to the UI thread before setting the "ItemsSource". Manipulating UI elements from another thread then the UI thread will cause runtime exceptions and must be avoided. We will revisit the concept of moving from worker thread to UI thread later in this tutorial when implementing the UI for the Android and iOS versions.

When running the Windows Phone 7 app it should list arrival and departures from Oslo airport.



6. Creating the UI skeleton

Now that we have the model and data logic in place we can start thinking about the UI of our mobile app. The app will be pretty simple, and consist of three tabs/sections; Arrivals, Departures and Airports. The user selects an airport from the Airports section, and the app automatically downloads Flights from that airport and populates the Arrivals and Departures sections. This mockup shows what the UI will look like.

6.1. Windows Phone 7 UI skeleton

On Windows Phone 7 we will use the "Pivot"-control to implement the three sections of the app.

1. Add a reference to "Microsoft.Phone.Controls" from the "FlightsNorway" project.

2. Open "MainPage.xaml" and remove all content from the "LayoutGrid".

```
<Grid x:Name="LayoutRoot" Background="Transparent">
</Grid>
```

3. Add a XML namespace declaration at the top of the "MainPage.xaml" file:

```
xmlns:controls="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone.Controls"
```

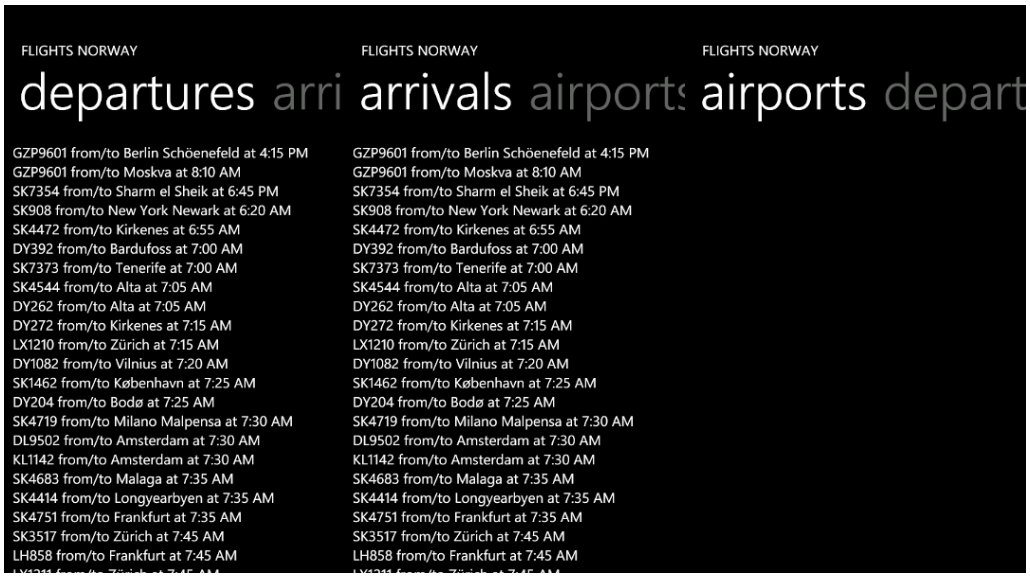
4. Add a "Pivot"-control inside the "LayoutGrid". Inside the "Pivot"-control you need to add three "PivotItem". Inside each "PivotItem" you need to add a "ListBox" to display data.

```
<controls:Pivot Title="FLIGHTS NORWAY">
    <controls:PivotItem Header="departures">
        <ListBox Name="_departures" Margin="0,0,-12,0" />
    </controls:PivotItem>

    <controls:PivotItem Header="arrivals">
        <ListBox Name="_arrivals" Margin="0,0,-12,0" />
    </controls:PivotItem>

    <controls:PivotItem Header="airports">
        <ListBox Name="_airports" Margin="0,0,-12,0" />
    </controls:PivotItem>
</controls:Pivot>
```

5. Change your "MainPage_Loaded" method to set the "ItemsSource" on arrivals and departures instead of airports (like we did previously in the tutorial).

```
private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    var flightsService = new FlightsService();
    flightsService.GetFlightsFrom(flights => Deployment.Current.Dispatcher.BeginInvoke(
        () =>
            {
                _arrivals.ItemsSource = flights.Value;
                _departures.ItemsSource = flights.Value;
            }), new Airport {Code = "OSL"});
}
```

6. Running the app should give you a result similar to this:



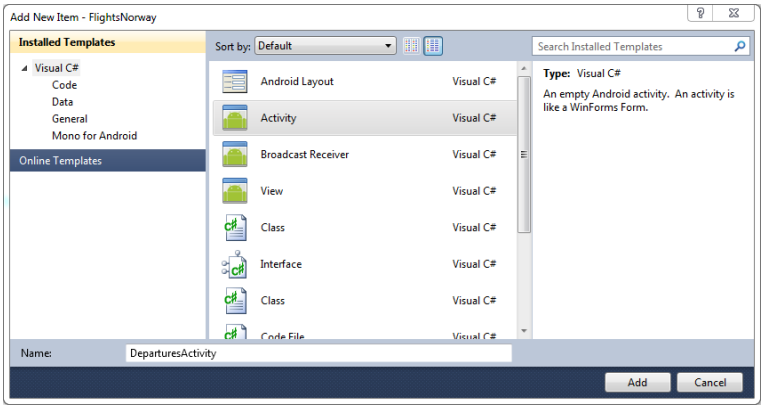
6.2. Mono for Android UI skeleton

For the Android version of the app we are going to use a "TabHost" to display the Departures, Arrivals and Airports tab.

1. Start by updating the "Main.axml"-file to contain a "TabHost".

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/tabhost"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <LinearLayout
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:padding="5dp">
    <TabWidget
        android:id="@android:id/tabs"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
    <FrameLayout
        android:id="@android:id/tabcontent"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:padding="5dp"/>
  </LinearLayout>
</TabHost>
```

2. Each of the tab items will be list activities. Right click the "FlightsNorway" Mono for Android project and select "Add new item", and then select the "Activity" template. Name the activity "ArrivalsActivity". Repeat this for "DeparturesActivity" and "AirportsActivity".



Change the "Label" property of the "Activity"-attribute to "FlightsNorway". The class declarations should look like this:

```
[Activity(Label = "FlightsNorway")]
public class DeparturesActivity : Activity
{
}
```

3. Next you need to download the icons that will be used for each of the tabs. The icons should be downloaded and added to the "Resources/Drawable"-folder with "BuildAction" set to "AndroidResource". The icons can be downloaded from:

   - https://github.com/follesoe/FlightsNorway/raw/master/MonoDroid/FlightsNorway/Resources/drawable/Departures.png
   - https://github.com/follesoe/FlightsNorway/raw/master/MonoDroid/FlightsNorway/Resources/drawable/Arrivals.png
   - https://github.com/follesoe/FlightsNorway/raw/master/MonoDroid/FlightsNorway/Resources/drawable/Airports.png

   The images are transparent PNG images, containing white graphics, so you will not be able to see the images in your web browser. But as long as you do not get a 404 you got the correct url and you can right click to save the image.

4. Now that we have the icons for the tabs we need to add a layout files for the tab icons. Add and XML file called "ic_tab_airports.xml" to the "Resources/Drawable"-folder.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:drawable="@drawable/airports" android:state_selected="true"/>
  <item android:drawable="@drawable/airports"/>
</selector>
```

5. Repeat the previous step and add two more files; "ic_tab_arrivals.xml" and "ic_tab_departures.xml". The only difference in the content of the file is the property of the "drawable" attribute. Make it "departures" in the "ic_tab_departures.xml"-file and "arrivals" in the "ic_tab_arrivals.xml"-file.

6. Earlier in this tutorial "Activity1" was deriving from "ListActivity". Change the class declaration and make it derive from "TabActivity".

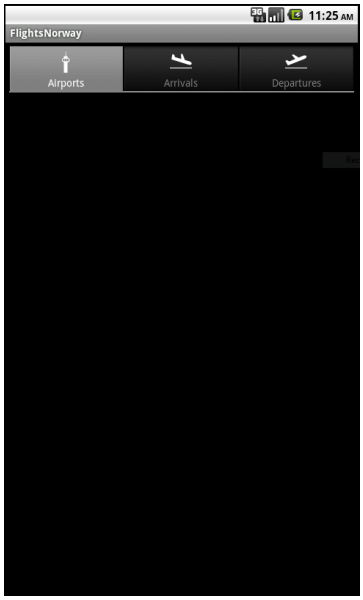
7. Next you need a small helper method to create each of the tabs hosting the "ArrivalsAcitivty", "DeparturesActivity" and "AirportsActivity" you just created.

```
private TabHost.TabSpec GetTab(string name, string title, Type type, int iconId)
{
    var intent = new Intent(this, type);
    intent.AddFlags(ActivityFlags.NewTask);


    var spec = TabHost.NewTabSpec(name);
    spec.SetIndicator(title, Resources.GetDrawable(iconId));
    spec.SetContent(intent);

    return spec;
}
```

8. Edit "OnCreate"-method of "Activity1". Make it add three tabs by calling the "GetTab"-method.

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    SetContentView(Resource.Layout.Main);

    TabHost.AddTab(GetTab("airports",
                          "Airports",
                          typeof(AirportsActivity),
                          Resource.Drawable.ic_tab_airports));

    TabHost.AddTab(GetTab("arrivals",
                          "Arrivals",
                          typeof(ArrivalsActivity),
                          Resource.Drawable.ic_tab_arrivals));

    TabHost.AddTab(GetTab("departures",
                          "Departures",
                          typeof(DeparturesActivity),
                          Resource.Drawable.ic_tab_departures));
}
```

If you run the app in the simulator it should look something like this:



6.3. MonoTouch UI skeleton

For the MonoTouch UI we are going to use a "UITabBarController" to host the Arrivals, Departures and Airports tabs. Each of the tabs will be extending "UITableViewController".

1. We already have the "AirportsTableViewController" from a previous step in this tutorial. Add two new classes to the "Tables"-folder, called "ArrivalsTableViewController" and "DeparturesTableViewController". The classes should derive from "UITableViewController". We will look at the implementation of the classes later in this tutorial. For now we only need them as place holders.

2. Next we need to add the controller responsible for managing the tabs. Add a new class called "MainTabBarController" to the root of the "FlightsNorway" project. Make the class derive from "UITabBarController". The class will create the tabs in the "ViewDidLoad"-method.

```
public override void ViewDidLoad ()
{
        base.ViewDidLoad();

        var tabs = new UIViewController[3];

        tabs[0] = new UINavigationController(new ArrivalsTableViewController());
        tabs[1] = new UINavigationController(new DeparturesTableViewController());
        tabs[2] = new UINavigationController(new AirportsTableViewController());

        tabs[0].TabBarItem =
          new UITabBarItem("Arrivals", UIImage.FromBundle("Content/Arrivals.png"), 1);

        tabs[1].TabBarItem =
          new UITabBarItem("Departures", UIImage.FromBundle("Content/Departures.png"), 1);

        tabs[2].TabBarItem =
          new UITabBarItem("Airports", UIImage.FromBundle("Content/Airports.png"), 1);

        ViewControllers = tabs;
}
```

3. Each of the tabs displays an icon. The icons can be downloaded from:

    - https://github.com/follesoe/FlightsNorway/raw/master/MonoTouch/FlightsNorway/Images/Airports.png
    - https://github.com/follesoe/FlightsNorway/raw/master/MonoTouch/FlightsNorway/Images/Airports@2x.png
    - https://github.com/follesoe/FlightsNorway/raw/master/MonoTouch/FlightsNorway/Images/Arrivals.png
    - https://github.com/follesoe/FlightsNorway/raw/master/MonoTouch/FlightsNorway/Images/Arrivals@2x.png
    - https://github.com/follesoe/FlightsNorway/raw/master/MonoTouch/FlightsNorway/Images/Departures.png
    - https://github.com/follesoe/FlightsNorway/raw/master/MonoTouch/FlightsNorway/Images/Departures@2x.png

    The images are transparent PNG images, containing white graphics, so you will not be able to see the images in your web browser. But as long as you do not get a 404 you got the correct url and you can right click to save the image.

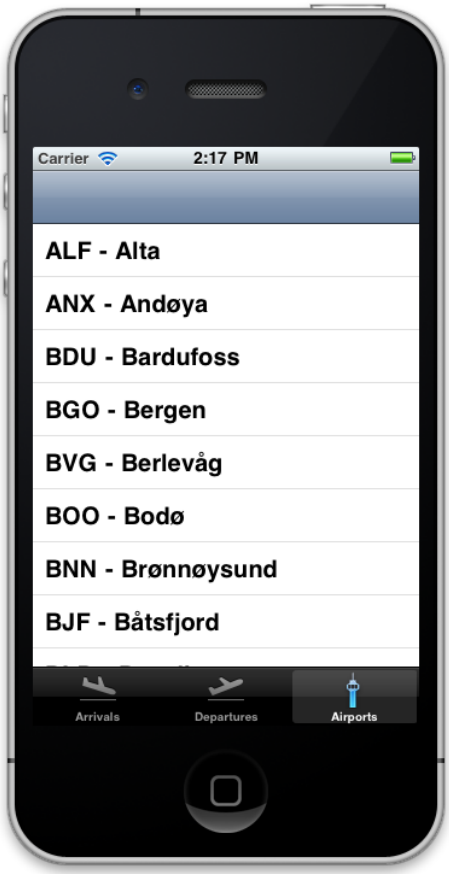
    The "@2x.png"-suffix version of the icons are used to provide high resolution versions of the icons for devices with higher resolution, like the iPhone 4 and the iPad.

    Add the icons to the "Content" folder, and set the "Build Action" to "Content".

4. Change the "AppDelegateIPhone" main class to create an instance of "MainTabBarController" in the "FinishedLaunching" method.

```
private MainTabBarController _mainTabs;

public override bool FinishedLaunching (UIApplication app, NSDictionary options)
{
        _mainTabs = new MainTabBarController();

        window.AddSubview(_mainTabs.View);
        window.MakeKeyAndVisible ();

        return true;
}
```

When you run the iPhone app and click the Airports tab it should look something like this:

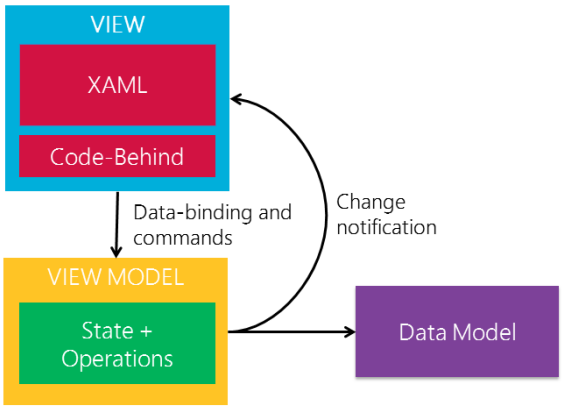


7. Implementing the MVVM pattern

7.1. About the MVVM pattern

Now that we have our REST clients implemented and a UI skeleton in place we are ready to populate the UI with real data. The most straight forward thing to do would be to write code accessing the REST clients directly in the XAML code-behind files for the Windows Phone 7 app, the "UITableViewController"-classes for the MonoTouch app, or in the "ListActivity"-classes for the Mono for Android app. This wouldn't be too bad, but it would lead to a fair amount of code being tied to the UI and not reusable across all three platforms. We can do better. In this next section of the tutorial we are going to apply the MVVM design pattern to get an even clearer separation of concerns in the UI layer.

The Model-View-View Model is a popular pattern among Silverlight, Windows Phone and WPF developers. The MVVM pattern is built around the concept of View Models, independent classes responsible of exposing the state and operations of the view. The View use data binding to synchronize with the View Model. The MVVM pattern belongs to the "Separated Presentation" family of patterns, and provides a clear separation of UI code (XAML) and the state of the UI (View Model). The View Model can be developed and tested independently of the View.



One of the added benefits of the MVVM pattern is that we can start to reuse code living close to the UI layer of the application across the different platforms. The View Model from a Windows Phone 7 app can take the role of the Model of the Model View Controller pattern used by the iOS and Android platform.

7.2. Airports View Model

The first View Model we are going to implement is the "AirportsViewModel". Every View Model implements the "INotifyPropertyChanged"-interface, so we are also going to implement a base class implementing this interface.

1. Start by adding a new folder to the "FlightsNorway.Lib"-project called "ViewModels".

2. Add a new class called "ViewModelBase" to the "ViewModels"-folder. The class is an abstract base class implementing the "INotifyPropertyChanged"-interface. The interface contains a single event called "PropertyChanged". This event is used to notify the view of any changes made to the View Model.

```
public abstract class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void RaisePropertyChanged(string propertyName)
    {
        var handler = PropertyChanged;

        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

3. Add a new class called "AirportsViewModel" to the "ViewModel"-folder. The View Model will expose two pieces of data, an "ObservableCollection<Airport>" containing a list of all the Norwegian airports the user can pick from, as well was a "SelectedAirport"-property keeping track of the currently selected Airport. The View Model will be responsible for calling the Airports service to parse the XML file with Airport names.

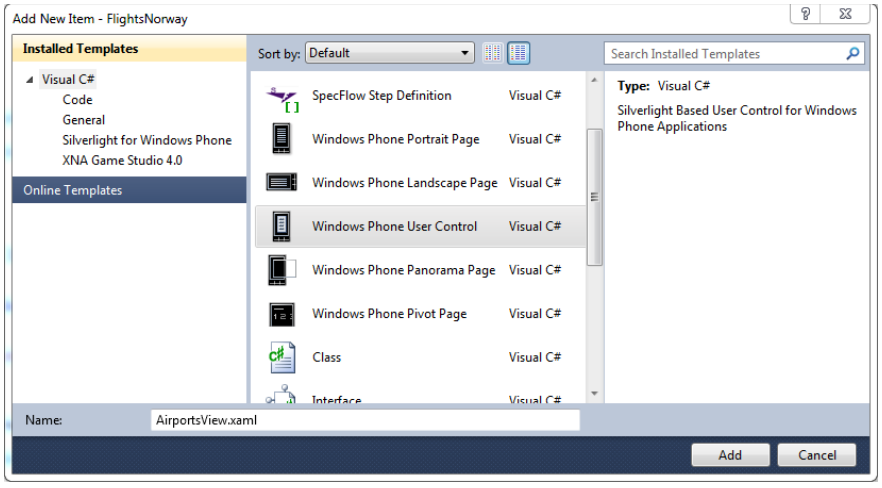
```
public class AirportsViewModel : ViewModelBase
{
    public ObservableCollection<Airport> Airports { get; private set; }

    private Airport _selectedAirport;

    public Airport SelectedAirport
    {
        get { return _selectedAirport; }
        set
        {
            if (_selectedAirport == value) return;

            _selectedAirport = value;
            RaisePropertyChanged("SelectedAirport");
        }
    }

    public AirportsViewModel(Stream norwegianAirports)
    {
        Airports = new ObservableCollection<Airport>();
        var service = new AirportNamesService();

        foreach(var airport in service.GetNorwegianAirports(norwegianAirports))
        {
            Airports.Add(airport);
        }
    }
}
```

7.3. Connecting the Airports View Model to the Windows Phone 7 app

Now that we have the Airports View Model implemented we can connect it to the Airports view. So far we've keep all the XAML code in the "MainPage.xaml" file. Now we are going to separate things a little bit by introducing a new "Views"-folder.

1. Add a "Views"-folder to the "FlightsNorway"-project.

2. Right click the "Views"-folder, and select "Add new item". Choose the "Windows Phone User Control" template.



3. Edit the "AirportsView.xaml"-file. The "LayoutRoot"-grid should look like this:

```
<Grid x:Name="LayoutRoot">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <ListBox Margin="8,8,8,8"/>
    <Button Grid.Row="1" Content="Save"/>
</Grid>
```

4. Edit the "MainPage.xaml"-file and add a new XML namespace declaration.

```
xmlns:Views="clr-namespace:FlightsNorway.Views"
```

5. Change the "Airports"-pivot element to hold the "AirpotsView" user control.

```
<controls:PivotItem Header="airports">
    <Views:AirportsView></Views:AirportsView>
</controls:PivotItem>
```

6. Edit the "AirportsView" code behind file to add an event handler for the "Loaded"-event. Inside the event handler you need to load the file containing Norwegian airports before creating the "AirportsViewModel" and assigning it to the "DataContext"-property of the user control. By assigning the "DataContext" property of the root object (the User Control) properties of the View Model will be bindable from any child element in the view.

```
public AirportsView()
{
    InitializeComponent();

    Loaded += AirportsView_Loaded;
}

private void AirportsView_Loaded(object sender, RoutedEventArgs e)
{
    var fileUri = new Uri("FlightsNorway;component/Content/Airports.xml",
                          UriKind.Relative);

    using (var stream = Application.GetResourceStream(fileUri).Stream)
    {
        DataContext = new AirportsViewModel(stream);
    }
}
```

7. The final step is to bind the "ItemsSource" and the "SelectedItem" property of the "ListBox"-control on the "AirportsView"-view. Edit the "AirportsView.xaml"-file to add the bindings. Setting the "SelectedAirport"-binding mode to "TwoWay" will synchronize the selection of an Airport back to the View Model.

```
<ListBox Margin="8,8,8,8"
           ItemsSource="{Binding Path=Airports}"
           SelectedItem="{Binding Path=SelectedAirport, Mode=TwoWay}" />
```

8. Run the Windows Phone 7 app to verify that you can select airports.

7.4. Connecting the Airports View Model to the Android app

Each of the tabs in the Android UI is created by deriving from "ListActivity". Earlier in this tutorial we made the main view display the list of Airports. This was implemented by setting the "ListAdapter" property of the "ListActivity" to an instance of an "ArrayAdapter<Airport>" adapter. Since we are going to reuse the View Model classes from the Windows Phone 7 app we are going to implement a custom "ObservableAdapter<T>", which can wrap any "ObservableCollection<T>" exposed by our View Model classes.

1. Add a new class called "ObservableAdapter" to the "FlightsNorway" Mono for Android project. The class declaration should look like this:

```
public class ObservableAdapter<T> : BaseAdapter<T>
{
}
```

2. Add a constructor to the class accepting the parent Activity, as well as an "ObservableCollection<T>" which the adapter is going to expose. You also need to add an event handler to the "CollectionChanged"-event. This event is fired every time an item is added or removed from the collection. When the event is fired we call the "NotifyDataSetChanged"-method on the "BaseAdapter"-class. This will notify the "ListActivity" using this adapter that the data was changed and that the UI needs to be updated.

```
private readonly Activity _context;
private readonly ObservableCollection<T> _collection;

public ObservableAdapter(Activity context, ObservableCollection<T> collection)
{
    _context = context;
    _collection = collection;
    _collection.CollectionChanged += (o, e) => NotifyDataSetChanged();
}
```

3. The "BaseAdapter"-class contains some abstract methods we need to implement. These methods are fairly straight forward, and can be implemented by simply accessing the "ObservableCollection" being adapted.

```
public override int Count
{
    get { return _collection.Count; }
}

public override T this[int position]
{
    get { return _collection[position]; }
}

public override long GetItemId(int position)
{
    return position;
}
```

4. To implement the "GetView"-method we first need to define the View. Add a new XML file called "list_item.xml" to the "Resources/Layout"-folder.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="wrap_content">
  <TextView xmlns:android="http://schemas.android.com/apk/res/android"
      android:id="@+id/text"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:padding="10dp"
      android:textSize="14sp">
  </TextView>
</LinearLayout>
```

5. The "GetView"-method gets and element from the "ObservableCollection", and creates a default one line text UI using the view we just defined. The text of the view is set to the "ToString"-representation of the item retrieved from the collection.

```
public override View GetView(int position, View convertView, ViewGroup parent)
{
    var item = _collection[position];
    var view = (convertView ??
        _context.LayoutInflater.Inflate(Resource.Layout.list_item, parent, false))
            as LinearLayout;

    var textView = (TextView)view.FindViewById(Resource.Id.text);
    textView.SetText(item.ToString(), TextView.BufferType.Normal);

    return view;
}
```

6. The final step is to update the "AirportsActivity"-class to derive from "ListActivity", create an instance of the "AirportsViewModel"-class, and pass it to the "ObservableAdapter" you just created. Since Mono for Android does not support the concept of data binding we have to manually synchronize changes back to the View Model. To do that you need to add an event listener to the "ItemClick"-event. In the event handler we set the "SelectedAirport"-property on the View Model.

```
[Activity(Label = "FlightsNorway")]
public class AirportsActivity : ListActivity
{
    private AirportsViewModel _viewModel;

    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);

        using(var stream = Assets.Open("Airports.xml"))
        {
            _viewModel = new AirportsViewModel(stream);
        }

        ListAdapter = new ObservableAdapter<Airport>(this, _viewModel.Airports);
        ListView.ItemClick += (o, e) =>
                _viewModel.SelectedAirport = _viewModel.Airports[e.Position];

    }
}
```

Run the Mono for Android app to verify that you can open the Airport tab and get see a list of airports.

7.5. Connecting the Airports View Model to the iOS app

Each of the tabs in the MonoTouch app is deriving from "UITableViewController", and each of the "UITableViewController"-classes has a "DataSource" set on the "UITableView". Just like we were able to create a generic "ObservableAdapter" in Mono for Android we will create a generic "ObservableDataSource" that we can use on all "UITableViewController" implementations.

1. Add a new class called "ObservableDataSource" to the "Tables" folder of the "FlightsNorway" MonoTouch project. The class declaration and constructor should look like this:

```
public abstract class ObservableDataSource<T> : UITableViewDataSource
{
        private ObservableCollection<T> _collection;
        private UITableView _tableView;

        public ObservableDataSource (ObservableCollection<T> collection,
                                     UITableView tableView)
        {
                _tableView = tableView;
                _collection = collection;
                _collection.CollectionChanged += (o, e) => {
                        tableView.ReloadData();
                };
        }
}
```

   The constructor accepts two parameters, the "ObservableCollection<T>" and the "UITableView. The constructor adds an event listener for the "CollectionChanged"-event that notifies the "UITableView" that the data has changed and the view needs to refresh itself.

2. The class exposes an abstract property called "CellID". This property is used by MonoTouch to identify "UITableViewCell"-objects so that they can be reused when they move off screen. The "RowsInSection"-method simply returns the number of items in the "ObservableCollection<T>".

```
public abstract NSString CellID { get; }

public override int RowsInSection (UITableView tableView, int section)
{
        return _collection.Count;
}
```

3. The "GetCell"-method is implemented to return a "UITableViewCell" for a given row. The method simply retrieves an object from the "ObservableCollection<T>" exposed by the data source class, and returns a single line cell displaying the "ToString"-representation of the object.

```
public override UITableViewCell GetCell (UITableView tableView, NSIndexPath indexPath)
{
    var cell = tableView.DequeueReusableCell(CellID);
    if (cell == null)
    {
        cell = new UITableViewCell(UITableViewCellStyle.Default, CellID);
                cell.TextLabel.Font = UIFont.FromName("Georgia", 16f);
    }

    cell.TextLabel.Text = _collection[indexPath.Row].ToString();
    return cell;
}
```

4. The "AirportsTableViewController" implementation needs to be updated to use the new "ObservableDataSource"-class. The "ViewDidLoad"-method should create an instance of the "AirportsViewModel", and set it to a private static field so that it can be accessed by the private "AirportsDataSource" and "AirportsDelegate" classes.

```
private static AirportsViewModel _viewModel;

public override void ViewDidLoad ()
{
        base.ViewDidLoad ();

        Title = "Airports";
        using(var stream = new FileStream("Content/Airports.xml", FileMode.Open))
        {
                _viewModel = new AirportsViewModel(stream);
        }
        TableView.DataSource = new AirportsDataSource(TableView);
        TableView.Delegate = new AirportsDelegate();
}
```

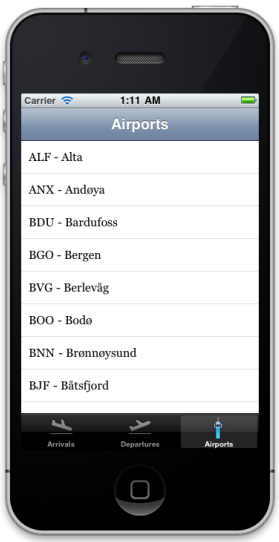
5. The private inner class "AirportsDataSource" also needs to be updated to derive from the "ObservableDataSource"-class. The class simply define the "CellID", as well as passing the Airports collection and the "UITableView" objects to the base class constructor.

```
private class AirportsDataSource : ObservableDataSource<Airport>
{
        private NSString _cellID = new NSString("Airports");
        public override NSString CellID { get { return _cellID; } }

        public AirportsDataSource(UITableView tableView) :
                                base(_viewModel.Airports, tableView)
        {
        }
}
```

6. In order to synchronize the selected Airport back to the View Model we need to add a "UITableViewDelegate"-class. This class is responsible for responding to user interaction with the table. The class implements the "RowSelected"-method and sets the "SelectedAirport"-property on the "AirportsViewModel".

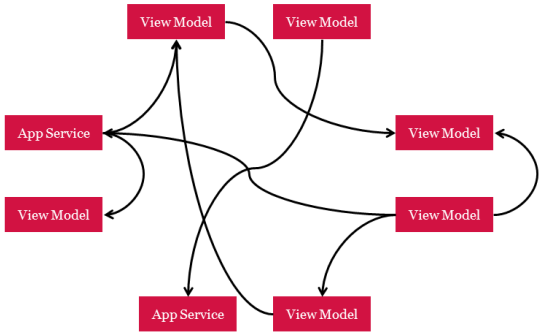
```
private class AirportsDelegate : UITableViewDelegate
{
        public override void RowSelected (UITableView tableView, NSIndexPath indexPath)
        {
                _viewModel.SelectedAirport = _viewModel.Airports[indexPath.Row];
        }
}
```

If you run the app and click the Airports tab the UI should look something like this:

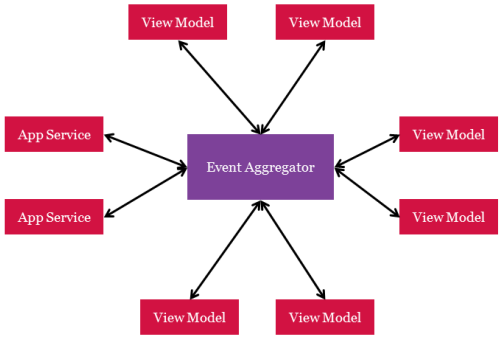


8. Passing messages between View Models using a Messenger

So far we have implemented the first View Model to display and select Airports. The natural next step is to implement the View Model to display Arrivals and Departures. When the user selects an Airport the Flights View Model needs to know about this so that it can refresh itself with flight information for the selected airport. There are multiple ways to implement this behavior. The most obvious way would be to have the "FlightsViewModel" reference the "AirportsViewModel". This can easily lead to a messy and inflexible design where the View Models are coupled to one another.



One god way to implement View Model collaboration is to use the Pub/Sub pattern, where all communication goes through a "man in the middle". In the context of MVVM this is often referred to as a Messenger/Event Aggregator.



To implement the Messenger pattern in the Flights Norway app we are going to use "TinyMessenger" from Steven Robbins. It's a complete publish/subscribe messenger contained in a single class that can be dropped into your project.

1. Download "TinyMessenger" from https://github.com/follesoe/FlightsNorway/raw/master/WindowsPhone/FlightsNorway.Lib/MVVM/TinyMessenger.cs and add it to the "FlightsNorway.Lib" in a new "Messages" folder.

2. Add a new class called "AirportSelectedMessage" to the "Messages"-folder. This message will be published when a user is done selecting which Airport to load Flights from.

```
public class AirportSelectedMessage : GenericTinyMessage<Airport>
{
    public AirportSelectedMessage(object sender, Airport content)
        : base(sender, content)
    {
    }
}
```

3. All View Models needs to access the same Messenger in order to pass messages between one another. If you are using an IoC container you can register the Messenger in the container in Singleton Scope. In this tutorial we are going to keep it simple and use a Service Locator for shared services. Add a new class called "ServiceLocator" to the root of the "FlightsNorway.Lib"-project.

```
public static class ServiceLocator
{
    public static ITinyMessengerHub Messenger { get; private set; }

    static ServiceLocator()
    {
        Messenger = new TinyMessengerHub();
    }
}
```

4. Next we want the "AirportsViewModel" to publish the "AirportSelectedMessage" when the user is done making a selection. To do this we are going to add a new "SaveSelection"-method that publishes the message.

```
public void SaveSelection()
{
    Servi!ceLocator.Messenger.Publish
        (new AirportSelectedMessage(this, SelectedAirport));
}
```

8.1. Publishing the AirportSelectedMessage form Windows Phone 7

To publish the message that a selection was made we need to call the "SaveSelection"-method when we consider the user done making a selection. For the Windows Phone 7 app we have a dedicated button to "Save" the selected airport.

1. Add a click handler to the button XAML code in the "AirportsView.xaml"-file.

```
<Button Grid.Row="1" Content="Save" Click="Button_Click"/>
```

2. The "Button_Click"-method simply casts the DataContext to a "AirportsViewModel" object before calling "SaveSelection".

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    ((AirportsViewModel)DataContext).SaveSelection();
}
```

8.2. Publishing the AirportSelectedMessage from Android

For the Android app we are going to listen for an event when the selected tab is changed. When this event occurs we check if the previously selected tab was the Airports tab. If so we need to invoke the method on the View Model that publishes the "AirportSelectedMessage".

1. Add a new "SaveSelection"-method to the "AirportsActivity" that calls through to the View Model.

```
public void SaveSelection()
{
    _viewModel.SaveSelection();
}
```

2. Add an event listener for the "TabChanged"-event at the end of the "OnCreate"-method of "Activity1".

```
TabHost.TabChanged += OnTabChanged;
```

3. Implement the "TabHost_TabChanged"-method and call the "SaveSelection"-method if the user is navigating away from the Airports tab.

```
private string currentTabId;

private void OnTabChanged(object sender, TabHost.TabChangeEventArgs e)
{
    if(string.IsNullOrEmpty(currentTabId) || currentTabId == "airports")
    {
        var airportsActivity = (AirportsActivity)
                                        LocalActivityManager.GetActivity("airports");

        airportsActivity.SaveSelection();
    }
    currentTabId = e.TabId;
}
```

8.3. Publishing the AirportSelectedMessage from iOS

For the iOS app we want to publish the "AirportSelectedMessage" when the user navigates away from the "Airports" tab. Cocoa Touch has a set of methods on the "UIViewController"-class you can override to run code when the before the View is displayed or hidden.

1. Override the "ViewWillDisappear"-method and call "SaveSelection" from the "AirportsTableViewController".

```
public override void ViewWillDisappear (bool animated)
{
        base.ViewWillDisappear (animated);
        _viewModel.SaveSelection();
}
```

2. Override the "ViewWillAppear"-method to set the selected row to the "SelectedAirport" property from the View Model.

```
public override void ViewWillAppear (bool animated)
{
        base.ViewWillAppear (animated);
        ((AirportsDataSource)TableView.DataSource).SetSelectedRow();
}
```

3. Add a "TableView"-property to the "ObservableDataSource" base class.

```
protected UITableView TableView { get { return _tableView; } }
```

4. Add a "SetSelectedRow"-method to the "AirportsDataSource" class.

```
public void SetSelectedRow()
{
        if(_viewModel.SelectedAirport != null)
        {
                int index = _viewModel.Airports.IndexOf(_viewModel.SelectedAirport);
                TableView.SelectRow(NSIndexPath.FromRowSection(index, 0),
                        false, UITableViewScrollPosition.None);
        }
}
```

8.4. Subscribing to the AirportSelectedMessage to update Title on iOS

To test that the message publishing works as expected we are going subscribe to the "AirportSelected"-message and update the title of the Arrivals and Departures tabs of the iOS app.

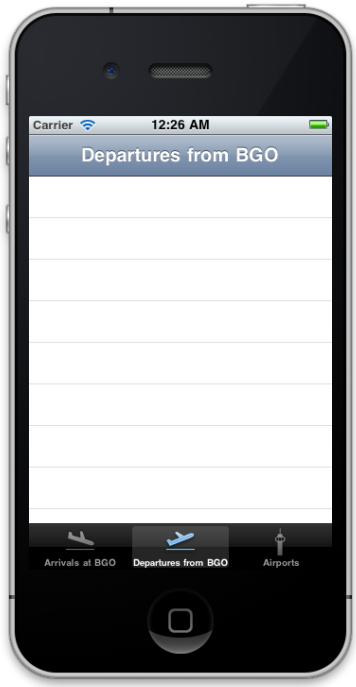
1. Subscribe to the "AirportSelectedMessage" in the "ArrivalsTableViewController" and set the Title to the selected Airport code.

```
public ArrivalsTableViewController ()
{
        ServiceLocator.Messenger.Subscribe<AirportSelectedMessage>(message => {
                Title = "Arrivals at " + message.Content.Code;
        });
}
```

2. Do the same for the "DeparturesTableViewController".

```
public DeparturesTableViewController ()
{
        ServiceLocator.Messenger.Subscribe<AirportSelectedMessage>(message => {
                Title = "Departures from " + message.Content.Code;
        });
}
```

Run the app and navigate to the Airports tab and make a selection. When you navigate to the Arrivals or Departures tab the title should reflect the selected airport.



9. Abstracting device specific functionality

From time to time you need to access functionality that exist across all three platforms, but is exposed by different APIs. Remember, neither MonoTouch nor Mon for Android provide abstractions on top of the underlying platform, instead they expose the actual APIs of the platform. One example of such functionality is invoking code on the UI thread. On MonoTouch we call the "InvokeOnMainThread"-method, on Mono for Android we use the "RunOnUiThread", while Windows Phone 7 provide this functionality in the "Dispatcher.BeginInvoke"-method.

We've previously seen how we can deal with minor differences in APIs using preprocessor directives. However, this approach easily violates the DRY (don't repeat yourself) principle if the same functionality is used multiple places in your code. You would need to duplicate the preprocessing directives and the platform specific API calls for ever situation you need to use the functionality. In this section we are going to see how we can abstract similar functionality behind a shared interface so that the calling code does not need to care about the platform it is running on.

9.1. Invoking code on UI thread on all three platforms

All three platforms expose APIs to dispatch a method call on the UI thread. This is typically done to switch context from a worker thread to the UI thread before running code that manipulates the UI. For our app this is relevant in the callback handlers for the REST services, as the call back is executing on the same worker thread that made the web request. Before we can manipulate the "ObservableCollections" (which in turn manipulates the UI), we need to move execution to the UI thread.

1. Start by adding a new interface to the root of the "FlightsNorway.Lib" project. Call the interface "IDispatchOnUIThread". The interface definition is simple and only consists of a single "Invoke"-method.

```
public interface IDispatchOnUIThread
{
    void Invoke(Action action);
}
```

2. Add a new class called "DispatchAdapter" to the "FlightsNorway" Windows Phone 7 app project.

```
public class DispatchAdapter : IDispatchOnUIThread
{
    public void Invoke(Action action)
    {
        Deployment.Current.Dispatcher.BeginInvoke(action);
    }
}
```

3. Add a "DisaptchAdapter"-class to the "FlightsNorway" Mono for Android project. The Mono for Android implementation needs a little bit more code, as it needs to hold a reference to an "Activity"-object in order to call the "RunOnUiThread"-method.

```
public class DispatchAdapter : IDispatchOnUIThread
{
    private readonly Activity _owner;

    public DispatchAdapter(Activity owner)
    {
        _owner = owner;
    }

    public void Invoke(Action action)
    {
        _owner.RunOnUiThread(action);
    }
}
```

4. Add a "DispatchAdapter"-class to the "FlightsNorway" MonoTouch project. The MonoTouch implementation is similar to the Mono for Android implementation.

```
public class DispatchAdapter : IDispatchOnUIThread
{
        private readonly NSObject _owner;

        public DispatchAdapter(NSObject owner)
        {
                _owner = owner;
        }

        public void Invoke (Action action)
        {
                _owner.BeginInvokeOnMainThread(new NSAction(action));
        }
}
```

5. Add a new property to the "ServiceLocator" to expose an instance of the "IDispatchOnUIThread"-interface. The property is static, and will be set to the correct implementation for each of the different platforms when the app starts.

```
public static IDispatchOnUIThread Dispatcher { get; set; }
```

6. In the constructor for "MainPage" in the Windows Phone 7 app set the "Dispatcher"-property to an instance of "DispatchAdapter".

```
public MainPage()
{
    InitializeComponent();
    ServiceLocator.Dispatcher = new DispatchAdapter();
    Loaded += MainPage_Loaded;
}
```

7. In the "OnCreate"-method of "Activity1" in the Mono for Android project set the "Dispatcher"-property to an instance of "DispatchAdapter".

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);

    ServiceLocator.Dispatcher = new DispatchAdapter(this);
    …
```

8. In the "FinishedLaunching"-method of "AppDelegateIPhone" for the MonoTouch project set the "Dispatcher"-property to an instance of "DispatchAdapter".

```
public override bool FinishedLaunching (UIApplication app, NSDictionary options)
{
        ServiceLocator.Dispatcher = new DispatchAdapter(this);
        …
```

Now that we have set the "IDispatchOnUIThread" to a platform specific implementation for each of the platforms the shared View Model code can simply use "ServiceLocator.Dispatcher.Invoke" to execute code on the UI thread. Each of the View Models does not have to care about platform specifics as these are abstracted behind a shared interface. This is a useful pattern when dealing with shared functionality exposed by different APIs across the three platforms.

10. Displaying Arrivals and Departures

10.1. Implementing the Flights View Model

Now that we have a way to select an airport and call the REST service to get flight information it's time to implement the Flights View Model. The View Model will listen for the "AirportSelectedMessage" and call the service when the message is received. The returned flight objects will be added to two "ObservableCollection<Flight>" holding Arrivals and Departures.

1. Add a new "FlightsViewModel"-class to the "ViewModels" folder of the "FlightsNorway.Lib"-project.

```
public class FlightsViewModel : ViewModelBase
{
    public ObservableCollection<Flight> Arrivals { get; private set; }
    public ObservableCollection<Flight> Departures { get; private set; }

    private FlightsService _service;

    public FlightsViewModel()
    {
        _service = new FlightsService();

        Arrivals = new ObservableCollection<Flight>();
        Departures = new ObservableCollection<Flight>();

        ServiceLocator.Messenger
            .Subscribe<AirportSelectedMessage>(m => LoadFlightsFrom(m.Content));
    }
}
```

2. When the "AirportSelectedMessage" is received the "LoadFlightsFrom"-method is invoked. This method is responsible for calling the REST service to retrieve Flights from the given Airport.

```
private void LoadFlightsFrom(Airport airport)
{
    _service.GetFlightsFrom(res => {
        if (!res.HasError())
        {
            ServiceLocator.Dispatcher.Invoke(() => AddFlights(res.Value));
        }
    }, airport);
}
```

3. When the callback for the "GetFlightsFrom"-method is invoked the dispatcher is used to invoke the "AddFlights"-method on the UI thread. The "AddFlights"-method is responsible for populating the Arrivals and Departures collections with flight objects.

```
private void AddFlights(IEnumerable<Flight> flights)
{
    Arrivals.Clear();
    Departures.Clear();

    foreach(var flight in flights)
    {
        if(flight.Direction == Direction.Arrival)
        {
            Arrivals.Add(flight);
        }
        else
        {
            Departures.Add(flight);
        }
    }
}
```

10.2. Displaying Arrivals and Departures on Windows Phone 7

To display arrivals and departures on Windows Phone 7 we need to add two new user controls to the "Views"-folder of the "FlightsNorway"-project.

1. Add a new item to the "Views"-folder of the "FlightsNorway"-project. Select the item type "Windows Phone 7 User Control" and name the file "ArrivalsView".

```
<Grid x:Name="LayoutRoot">
    <ListBox ItemsSource="{Binding Path=Arrivals}" />
</Grid>
```

2. Add a new view called "DeparturesView".

```
<Grid x:Name="LayoutRoot">
    <ListBox ItemsSource="{Binding Path=Arrivals}" />
</Grid>
```

3. Update the "MainPage.xaml" to use the newly created user controls inside the Pivot items.

```
<controls:PivotItem Header="departures">
    <Views:DeparturesView x:Name="_departuresView" />
</controls:PivotItem>

<controls:PivotItem Header="arrivals">
    <Views:ArrivalsView x:Name="_arrivalsView" />
</controls:PivotItem>
```

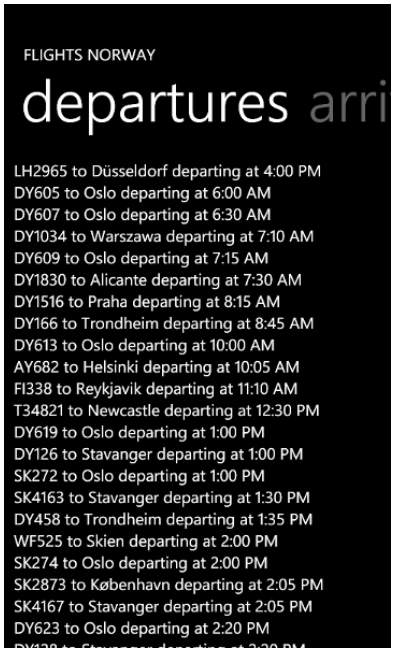
4. Change the "MainPage_Loaded"-method to create the "FlightsViewModel" and set the "DataContext" of the Arrivals view and the Departures view.

```
private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    var viewModel = new FlightsViewModel();
    _arrivalsView.DataContext = viewModel;
    _departuresView.DataContext = viewModel;
}
```

5. To make the UI more accurate you need to update the "ToString"-method of the Flight class. Currently it displays the same string for both arrivals and departures. You should return a string indicating if the flight is an arrival or departure.

```
public override string ToString()
{
    if(Direction == Direction.Arrival)
    {
        return
            string.Format("{0} from {1} arriving at {2}", FlightId, Airport.Name,
                                ScheduledTime.ToShortTimeString());
    }
    else
    {
        return
            string.Format("{0} to {1} departing at {2}", FlightId, Airport.Name,
                                ScheduledTime.ToShortTimeString());
    }
}
```

Running the Windows Phone 7 app, selecting an Airport and then navigating to the Departures pivot item should give you something like this:



10.3. Displaying Arrivals and Departures on Android

The "ArrivalsActivity" and "DeparturesActivity" will be responsible for displaying arrivals and departures in the Android app. They both need to share the same "FlightsViewModel"-instance. To keep this single we will expose the view model as a static property on "Activity1" that "ArrivalsActivity" and "DeparturesActivity" can access.

1. Add a new "FlightsViewModel" property to the "Activity1"-class.

```
public static FlightsViewModel FlightsViewModel { get; private set; }

protected override void OnCreate(Bundle bundle)
{
        base.OnCreate(bundle);

        FlightsViewModel = new FlightsViewModel();
        …
```

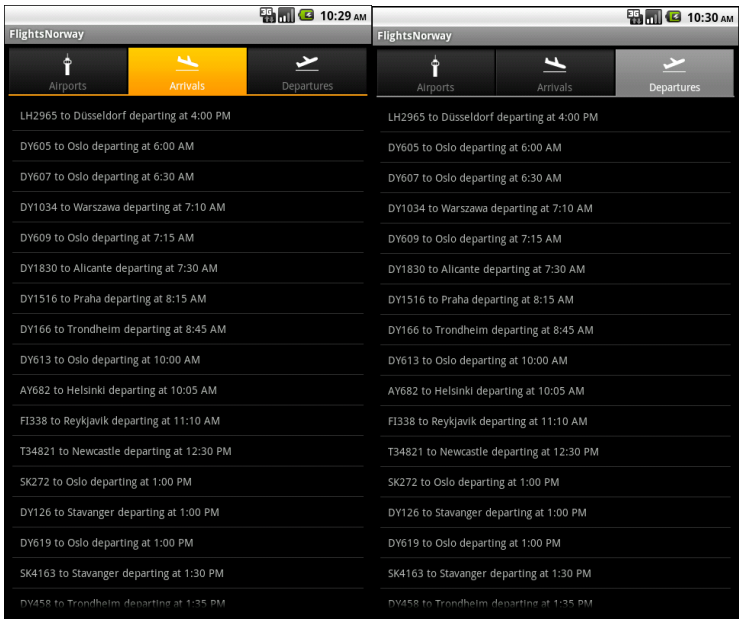
2. Edit the "ArrivalsActivity" and make it derive from "ListActivity" and override the "OnCreate"-method to set the "ObservableAdapter" as the "ListAdapter" for the activity.

```
[Activity(Label = "FlightsNorway")]
public class ArrivalsActivity : ListActivity
{
    protected override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);
        ListAdapter = new ObservableAdapter<Flight>
            (this, Activity1.FlightsViewModel.Departures);
    }
}
```

3. Edit the "DeparturesActivity" and make it derive from "ListActivity" and override the "OnCreate"-method to set the "ListAdapter".

```
[Activity(Label = "FlightsNorway")]
public class DeparturesActivity : ListActivity
{
    protected override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);
        ListAdapter = new ObservableAdapter<Flight>
            (this, Activity1.FlightsViewModel.Departures);
    }
}
```

Running the Android app, selecting an Airport, and then clicking the "Departures"/"Arrivals" tab should show something like this:



10.4. Displaying Arrivals and Departures on iOS

To display arrivals and departures on the iOS app we need to update the "ArrivalsTableViewController" and the "DeparturesTableViewController".

1. Both controllers need to share the instance of "FlightsViewModel", so it will be passed in as a constructor parameter. Make the same change for both "ArrivalsTableViewController" and "DeparturesTableViewController".

```
private readonly FlightsViewModel _viewModel;

public ArrivalsTableViewController (FlightsViewModel viewModel)
{
        _viewModel = viewModel;
        ...
```

2. Create a new private class called "ArrivalsDataSource" and set it as the "DataSource"-property of the "TableView".

```
public override void ViewDidLoad ()
{
        base.ViewDidLoad ();
        TableView.DataSource = new ArrivalsDataSource(_viewModel, TableView);
}

private class ArrivalsDataSource : ObservableDataSource<Flight>
{
        private NSString _cellID = new NSString("Arrivals");

        public override NSString CellID { get { return _cellID; } }

        public ArrivalsDataSource(FlightsViewModel viewModel, UITableView tableView) :
                base(viewModel.Arrivals, tableView)
        {
        }
}
```

3. Create a similar "DeparturesDataSource" class for the "DeparturesTableViewController". Make sure to set it in the "ViewDidLoad"-method.

```
private class DeparturesDataSource : ObservableDataSource<Flight>
{
        private NSString _cellID = new NSString("Departures");

        public override NSString CellID { get { return _cellID; } }

        public DeparturesDataSource(FlightsViewModel viewModel, UITableView tableView) :
                base(viewModel.Departures, tableView)
        {
        }
}
```

4. Update the "MainTabBarController" to pass in an instance of the "FlightsViewModel" to the Arrivals and Departures controllers.

```
var viewModel = new FlightsViewModel();
tabs[0] = new UINavigationController(new ArrivalsTableViewController(viewModel));
tabs[1] = new UINavigationController(new DeparturesTableViewController(viewModel));
```

Running the app, selecting an airport and navigating to Arrivals or Departures should give you:

11. Summary

Even though the focus of this tutorial has been on mobile apps, most of the "FlightsNorway.Lib" code is reusable in other contexts as well. You could easily build a mobile web app based on the same code, or a WPF or Silverlight based desktop app.

12. Appendix

12.1. Appendix 1 – Assemblies available on different platforms

Assembly	MonoTouch	Mono for Android	Windows Phone 7
`Mscorlib`	Yes	Yes	Yes
`System`	Yes	Yes	Yes
`System.Core`	Yes	Yes	Yes
`System.Data`	Yes	Yes	No
`System.Json`	Yes	Yes	No
`System.Runtime.Serialzation`	Yes	Yes	Yes
`System.ServiceModel`	Yes	Yes	Yes
`System.ServiceModel.Web`	Yes	Yes	Yes
`System.Transactions`	Yes	Yes	No
`System.Web.Services`	Yes	Yes	No
`System.Xml`	Yes	Yes	Yes
`System.Xml.Linq`	Yes	Yes	Yes
`System.Device`	No	No	Yes
`System.Observable`	No	No	Yes
`System.Windows`	No	No	Yes