
Progress
<Response>
https://bs3.cdn.telerik.com/v1/lmplloffsq4t2dgt/59fcedc0-b071-11e6-8748-ef6880f49a5c
</AircraftImageURL>
<CreatedAt>2016-11-22T05:17:22.090Z</CreatedAt>
<ModifiedAt>2016-11-22T05:17:22.090Z</ModifiedAt>
<CreatedBy>00000000-0000-0000-0000-000000000000</CreatedBy>
<ModifiedBy>00000000-0000-0000-0000-000000000000</ModifiedBy>
<Owner>00000000-0000-0000-0000-000000000000</Owner>
<Id>f265d8a0-b072-11e6-9683-4f3a0a880ceb</Id>
<Meta>
<Permissions>
<CanRead>true</CanRead>
<CanUpdate>true</CanUpdate>
<CanDelete>true</CanDelete>
</Permissions>
</Meta>
</result>
<result>...</result>
<result>...</result>
<result>...</result>
<result>...</result>
<result>...</result>
REAL-WORLD XAMARIN PROBLEMS
And How To Solve Them

Mobile Strategy

It's 2017. Developers have been making mobile apps for almost a decade and have some choice in how they build those apps. Whatever the approach, a mobile strategy is imperative—it's how you achieve consistency in building and supporting mobile solutions.

The choice of technology for mobile depends on various factors, including the skills of the team involved, app specifics, customer demographics and maintainability of code base. Here are the high level technology choices that could make up a mobile strategy:

Mobile Web

- Web apps that work nicely on mobile devices
- Frameworks like Twitter Bootstrap & Kendo UI Core help.

Native Apps

- Native iOS/Android/Windows apps for each platform
- Written in native languages and built with native toolkits
- Closest to the metal and best possible UX
- Expensive to write and maintain

JS Native Apps

- Use web technologies to build truly native mobile apps
- Cross platform reach with single codebase
- Big contenders being NativeScript and React Native
- Potential to share code between web and mobile

Cross-Compiled

- Use beloved language and tools to write mobile app
- Single codebase with cross-platform reach
- App gets compiled down to native code on each platform
- Primary contender is Xamarin

Why Xamarin?

If your development experience includes .NET, you probably already know C# and XAML—the primary programming languages for mobile development with Xamarin. Xamarin truly democratizes cross-platform mobile development for .NET developers. Here are some benefits you gain by taking the Xamarin route:

- Target iOS, Android, MacOS and UWP from single code base
- Truly native cross-platform apps
- Reach smart watches and smart TVs
- Reuse existing skills in C#/XAML
- Consistent cross-platform APIs for development
- Top notch familiar tooling/IDEs on Windows or Mac
- Support for full DevOps lifecycle
- Fantastic developer community

Get Your Ammunition

Now that you've decided on Xamarin as your chosen technology to build cross-platform native mobile apps, it's time to get all the ammunition needed to jumpstart your app development. Let's take a look at some of the moving pieces and essential tooling that will set you up for success with Xamarin development:

1. Start Right

Begin your Xamarin journey @ http://xamarin.com - a treasure trove of content that explains the promise of Xamarin and walks you through all the tooling. You'll want to be sure to understand the difference between Xamarin.iOS/Android and Xamarin.Forms —there are specific app cases where one approach may fit better.

2. Get the IDEs

When it comes to Xamarin development, you get two rich Integrated Development Environments (IDEs) to choose from:

- Visual Studio for Windows
- Xamarin Studio for Mac

Both IDEs suppport some common features for ease of development:

- Rich Intellisense support for code completion
- Intelligent typing assist for iOS/Android API mappings
- Built-in Xamarin project templates
- Easy code navigation and IDE configurations
- Switchable light and dark themes
- Choice of various device emulators with varying screen sizes
- Easy debug options for both emulators/device deployment
- Publish apps to stores directly from the IDE
- Integrated version control
- Robust User Interface Designers for iOS/Android

Both Visual Studio and Xamarin Studio come in completely free yet feature-rich Community editions. You do have to keep a few limitations in mind though:

- Visual Studio on Windows will need a Mac to act as an iOS build host running XCode—this is an Apple licensing requirement. The Mac could be hard wired to the Windows development machine somewhere on the network or in the cloud. In fact, with the recent iOS Windows Simulator, you don't ever have to leave the comforts of Visual Studio—the simulator will stream the iOS emulator from the Mac.
- Xamarin Studio on Mac can target iOS, Android and Mac, but not Windows. The app solution from Xamarin Studio could, however, be brought over to Visual Studio on a Windows development machine later to add a Windows-supporting project, while still leveraging the common Portable Class Library (PCL) for code sharing between platforms.

- The IDEs allow you the option to install device simulators for your chosen platforms—like iOS, Android and UWP. These simulators will likely depend on underlying platform SDKs and have heavy tooling dependencies—like Android SDKs and XCode—as well as often needing specialized hardware virtualization. And as mobile platforms move forward, there will be lots of updates across various SDKs, device simulators and tools. Xamarin developers are best advised to keep all development tools updated and OS patches installed from stable channels.

A new kid in the IDE block is Visual Studio for Mac. It's still in Preview, but you can clearly see the promise. For now, your development experience is about the same on Xamarin Studio and Visual Studio for Mac but the future will be interesting.

3. Leverage Open Source

As if the Microsoft acquisition of Xamarin wasn't a big enough deal, sweeter news greeted .NET developers in early 2016: Xamarin was going open source! All of the Xamarin frameworks, namely Xamarin.iOS, Xamarin.Android and Xamarin.Forms are completely open source, as are all the SDKs under a broad MIT license. Everything starts at open.xamarin.com.

The open sourcing of Xamarin has two major repercussions. First is that Xamarin and all of its tooling are now completely free. The second is that there is a true collaborative open source community being now built around Xamarin. You can be a part of it.

Normally, you would just download the Xamarin SDKs and start building your app. But the true developer in you wants to look under the covers—what is the framework doing for you? Now you can look through Xamarin's source code, pull it down and build the SDK/libraries yourself.

If you've been brave enough to look into Xamarin source code and build the tooling/frameworks locally, take the next logical step forward and contribute back. Do you think you can tweak something to be better, or fix a bug, or take a stab at a pending to-do item? This is your opportunity to pay back some technical debt and make the world a better place by contributing to open source projects. Since Xamarin empowers millions of developers there is a process to contributing:

- File a bug and track issues at https://bugzilla.xamarin.com/newbug.
- For design discussions, subscribe to forms-devel@lists.xamarin.com or macios-devel@lists.xamarin.com.
- Discuss issues in the Gitter chat room.
- Make appropriate code changes following the coding patterns, sign the .NET Foundation's CLA and make a pull request. The rest they say is for legends!

4. Get Essential Tooling

You may want to keep a few tools in your arsenal. These aren't critical, but often make your life easier:

- NuGet – As the de facto package manager for the .NET ecosystem, NuGet will faithfully serve your Xamarin projects as well. Bring in your favorite .NET libraries, services and utility wrappers.
- Xamarin Component Store – Your Xamarin app needs awesomeness and you don't need to reinvent the wheel. Augment your app with polished UI components, service SDKs and various cloud/social integrations all from one component store.
- Postman – As with most modern mobile apps, your Xamarin app is likely to use cloud services and APIs. With Postman, you get a powerful GUI platform to make your API development faster and easier, from building API requests through testing, documentation and sharing. The app for free for Windows, Mac, Linux or Chrome.

- Reflector – Want to collaborate with others or demo your Xamarin app? Reflector is a wireless mirroring and streaming receiver that works great with Google Cast™, AirPlay™ and AirParrot 2®. You get to mirror your content to the big screen without wires or complicated setups. You can play games, watch movies, demo applications or present from the palm of your hand.

- Fiddler – Want insight as to what's happening over the wire? Leverage the ever-popular Fiddler—the free web debugging proxy for any browser, system or platform. Did you know that in addition to Windows, we now have Fiddler Betas for Mac and Linux? Choice is a good thing for developers.

5. Use Helpful Utilities

As your app complexity grows, your Xamarin development environment and code base calls for sanity. A few frameworks and tools can help:

- MVVM Light – Bring some structure and sanity to your Xamarin apps by following the established Model-View-ViewModel (MVVM) pattern and including the MVVM Light framework. You get out-of-the-box features like commanding, messenger, inversion of control (IoC) and INotifyPropertyChanged UI data binding implementations.

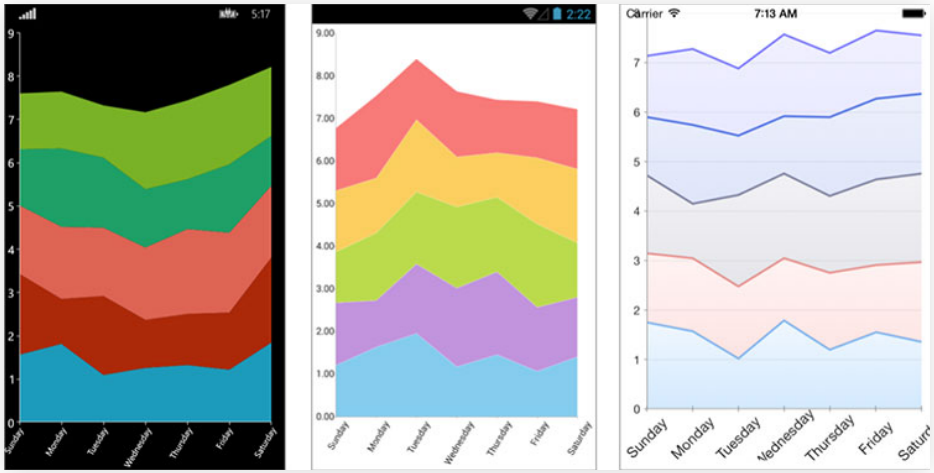
- Prism for Xamarin.Forms – Have you used and liked Prism for past XAML development? Now you can bring the Prism fun to Xamarin.Forms development with a NuGet package and an optional Prism Template pack to get you started. You get to leverage built-in MVVM features like commanding, ViewModelLocator, event aggregator, navigation and IoC with Unity.

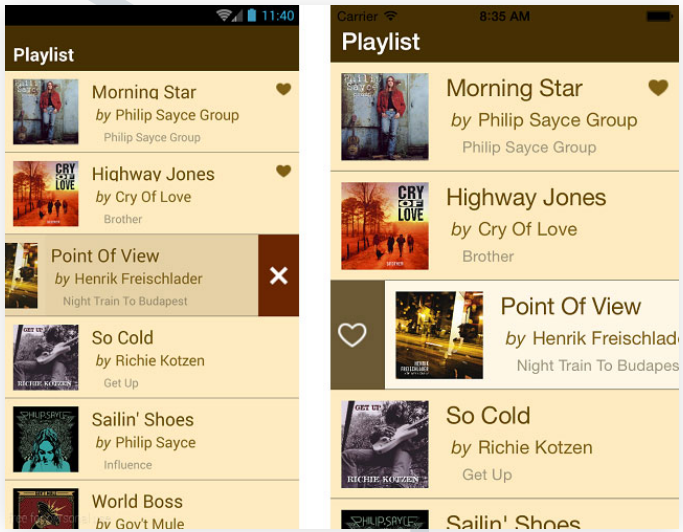
- Xamarin WorkBooks – Need a playground as you are building your Xamarin apps? Like experimenting with native API features, want to document a functionality for someone or simply see your tentative code in action? You're in luck with the newly introduced Xamarin Workbooks—a standalone application that allows for building interactive documents with executable live code outside of any apps. You could run code-fenced C# blocks inline or inside iOS/Android simulators. It's perfect to try out or share new features before adding them to your app.

- XUnit – This is the perfect open source unit testing framework for your Xamarin apps, for both PCLs and device-specific projects. Simply get a NuGet package in your project, write some unit tests and run them against your app, either through the command line, MSBuild or using the iOS/Android test runners.

- Xamarin Test Cloud – Your professional Xamarin apps shouldn't be at the mercy of mobile device idiosyncrasies. And let's face it, there are a plethora of devices across various platforms that may run your Xamarin app. The solution is Xamarin Test Cloud. You can automate app testing on 2000+ real devices, all running in the cloud. Increase confidence in your app by testing user interactions on real devices to find bugs, with complete memory and performance analysis. Get polished reports, fix problems and repeat—that's how you ship high quality apps.

- PreBuilt Apps – Jumpstart your Xamarin app development with a polished pre-built app—showcase apps with open source code in GitHub. Peruse unique features or incorporate UI components to give your Xamarin apps a flying start.





6. The Need For Polished UI

You've started building your dream Xamarin app, picked the IDE of your choice and grabbed the necessary frameworks. Guess what? Your end users do not see any of magic behind the scenes. What they do see is the app UI and the fluidity of user experience that your app provides. You can try reinventing the wheel on UI or go for some well-engineered UI controls out of the box.

Telerik® UI for Xamarin includes elegant, polished and performant native UI widgets for all your Xamarin apps. With UI for Xamarin, you'll get to deliver your Xamarin apps quicker and delight users with beautiful functional UI. What you get out of the box are UI widgets that are difficult to create by hand—like variety of Charts, polished ListView, handy SideDrawer, effective DataForm and much more.

There are various ways in which you could get started with Telerik UI for Xamarin or incorporate the bits in your existing Xamarin project: Go the NuGet route, use the Telerik Control Panel or download from the product page. The goals are to augment your Xamarin toolset with ease, provide rich performant UI out of the box, ship your app faster and delight your users.

Common Xamarin.Forms Developer Tips

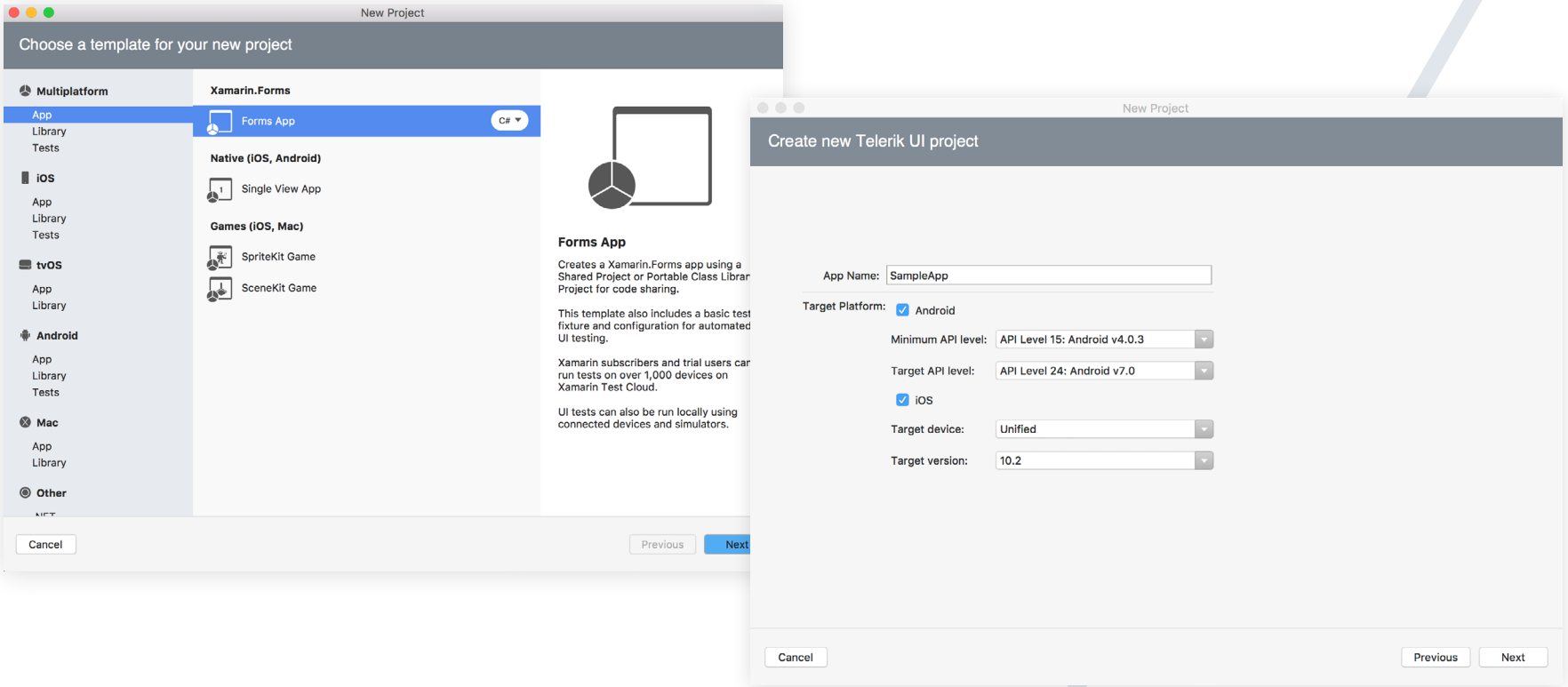
Are you building your next native cross-platform mobile app using Xamarin.Forms? Good for you! You get the benefits of a single C#/XAML codebase that targets all mobile platforms and customizes the user experience on each. Not only do you get to share business logic in C#, but you benefit from an abstracted shared UI layer as well—all in XAML, rendering native UI on each platform.

As you start building your app out though, there are some very real roadblocks to overcome before your app gets functional. You need to organize your app content for optimal usage and preserve the continuity of user experience. You need to have a consistent coding strategy on a couple of fronts and minimize resource usage. Here are some common developer roadblocks and practical tips when building apps using Xamarin.Forms, all in the context of a simple Aviation app that shows information about business and supersonic jets. All the code is open sourced ... but let's start breaking things down pragmatically.

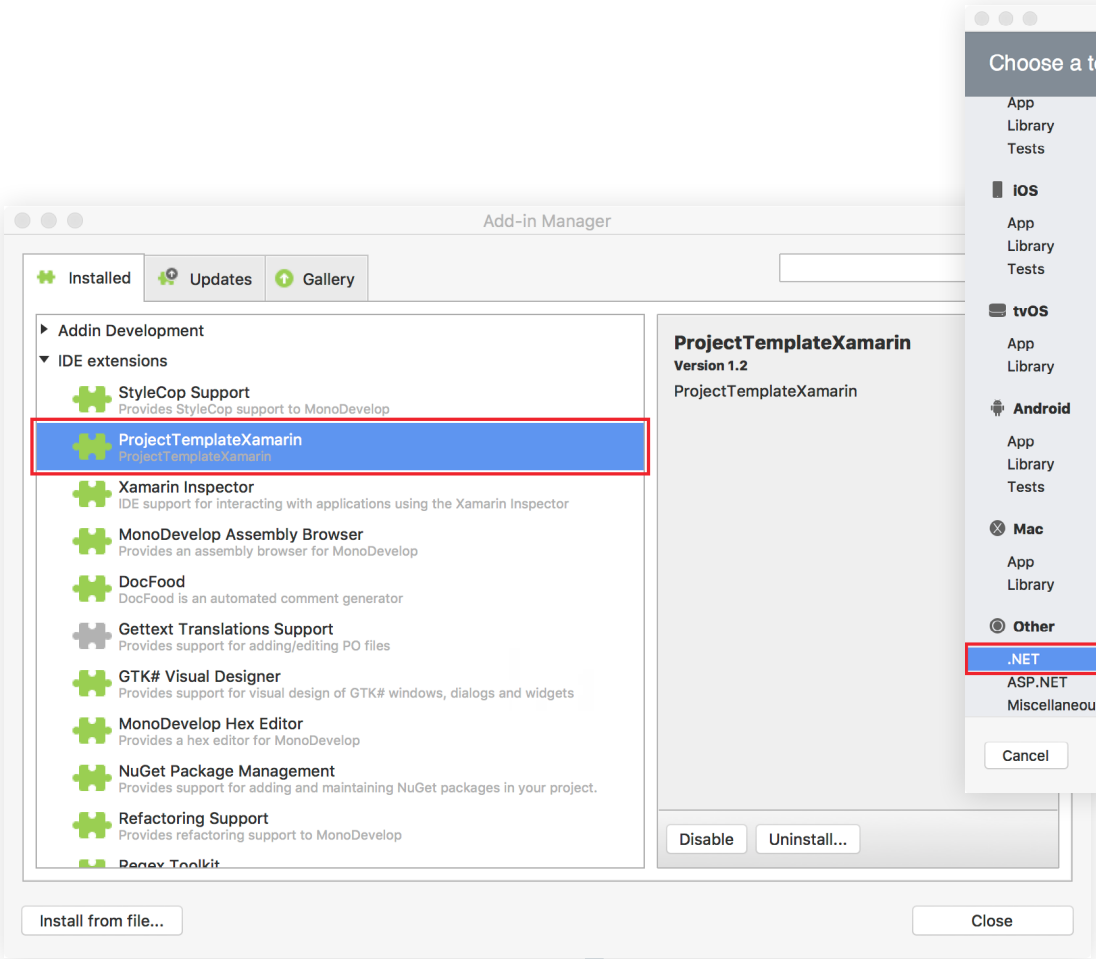
1. Begin Smart

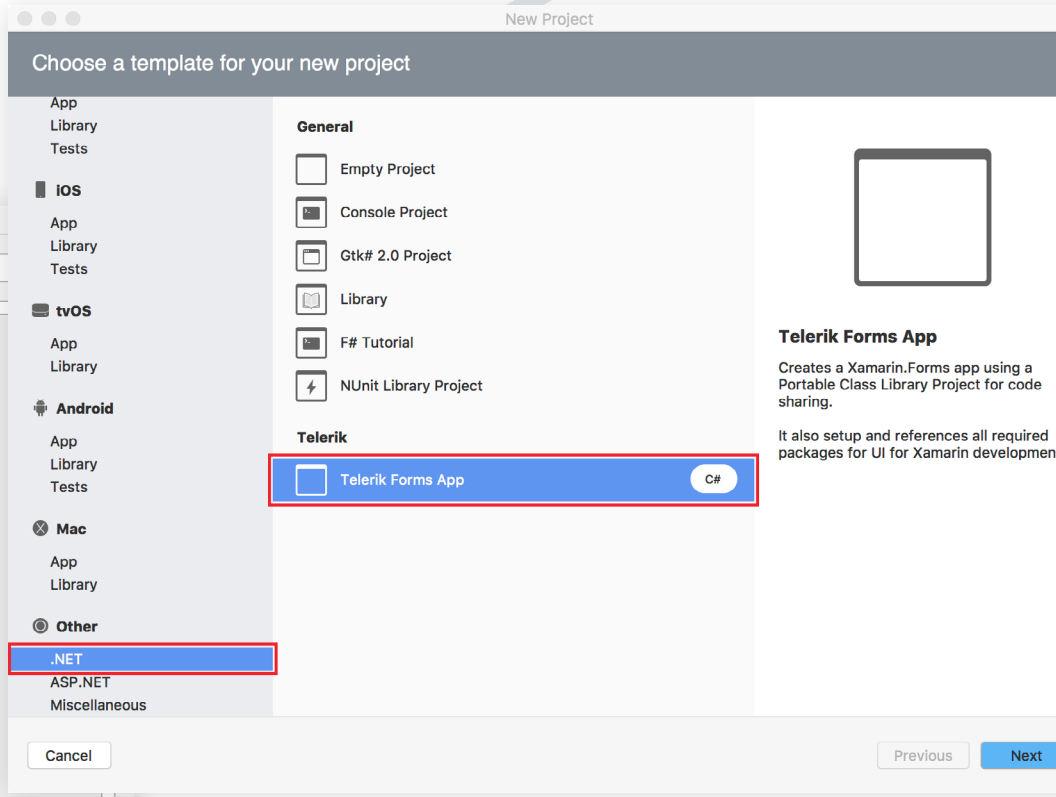
As you begin your greenfield Xamarin.Forms project, first take a good look at the project templates both in Visual Studio and Xamarin Studio. These templates give you a great starting point. You get to pick your choice of app platforms to target and mobile device API levels to support, and the end result is a project that is catered just for your needs.

The templates you get with Visual Studio and Xamarin Studio are identical—they let you choose whether you want to build a Xamarin. Forms app vs Xamarin.iOS/Android, or more specialized platform targets like MacOS, smart watches or smart TVs. Once you pick your platform, a wizard follows up with API level selections—pretty simple and lets you decide your strategy upfront.

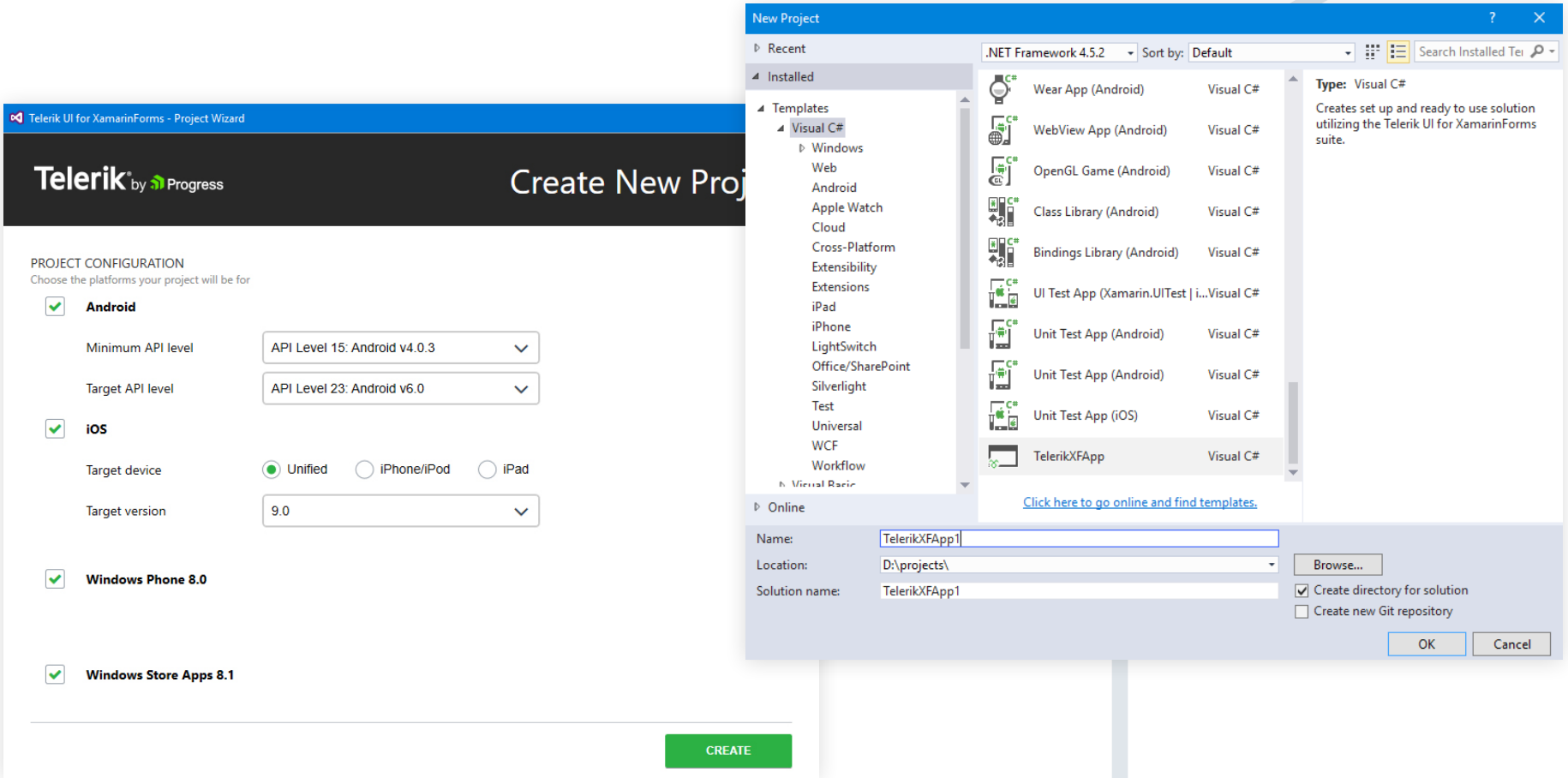


If you're using Telerik UI for Xamarin, we want to help you get started in the right direction. Accordingly, you get some additional Templates—VSIX files for Visual Studio and MPack for Xamarin Studio. Add them to your IDE and the templates light up for you, like in Xamarin Studio below:





You get a similar experience in Visual Studio, as below. These templates will add all the needed Telerik references to each of your projects—platform specific ones as well as shared Portable Class Libraries.

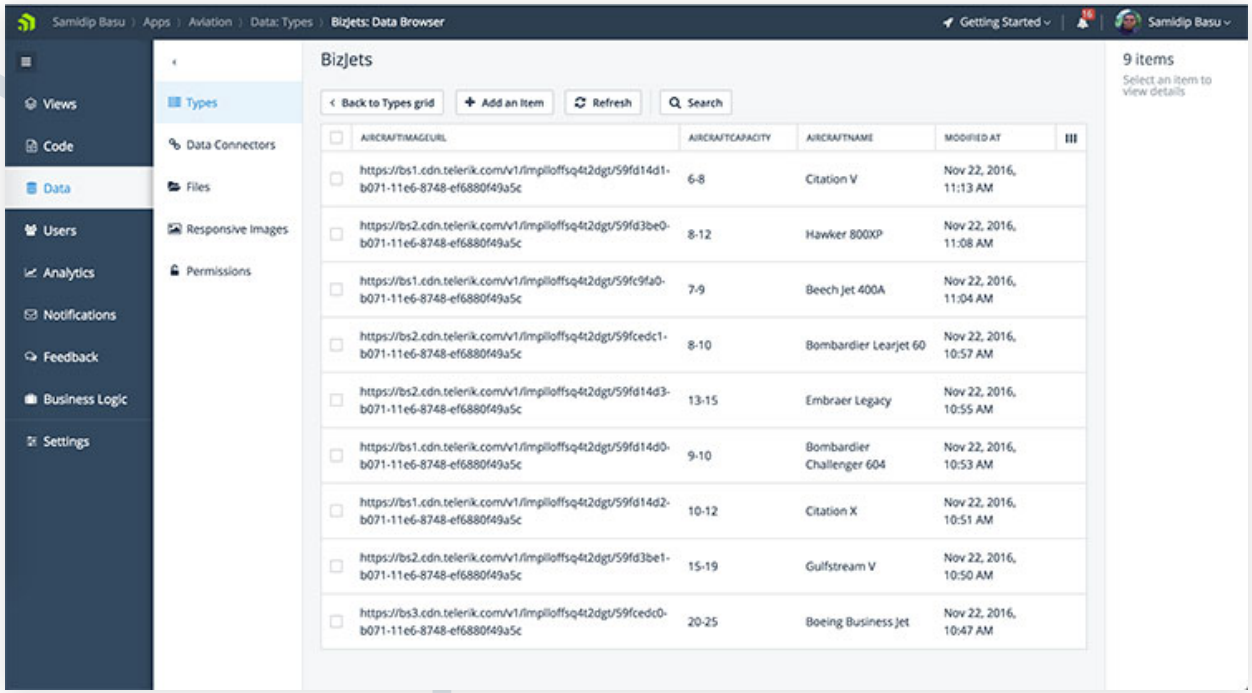


2. Have a Consistent Cloud Strategy

Data is central to any mobile app—it's the crux of what your app is all about. So it's no surprise that you have to strategize about how you manage, interact and present your data. In this day and age, most mobile users utilize a variety of mobile platforms and roam between various devices throughout the day. You want your app to follow the user and provide continuity of experiences.

Enter the cloud. Putting your app data in the cloud is a great way to have it be accessible from any app—mobile, web or desktop. Almost any cloud provider will give you the same benefits with data hosting—a RESTful service you can easily invoke from your Xamarin.Forms app. But maybe moving your data to the cloud isn't an option for you, and that is totally fine. You could host your data in a SQL Server running under your desk. The important part is to expose the data out as a service so it can be consumed easily from any platform.

The data to power the Aviation app sits in Telerik Backend Services—a reliable backend companion to mobile/web apps, with features like Push Notifications, User Authentication, Cloud Code and Responsive Images. The data sits in a simple table that lists all the available business jets. The pertinent pieces of information are the aircraft name, passenger capacity and URL to its corresponding image. Think of this as a SQL Server table—just running in the cloud with bells and whistles.



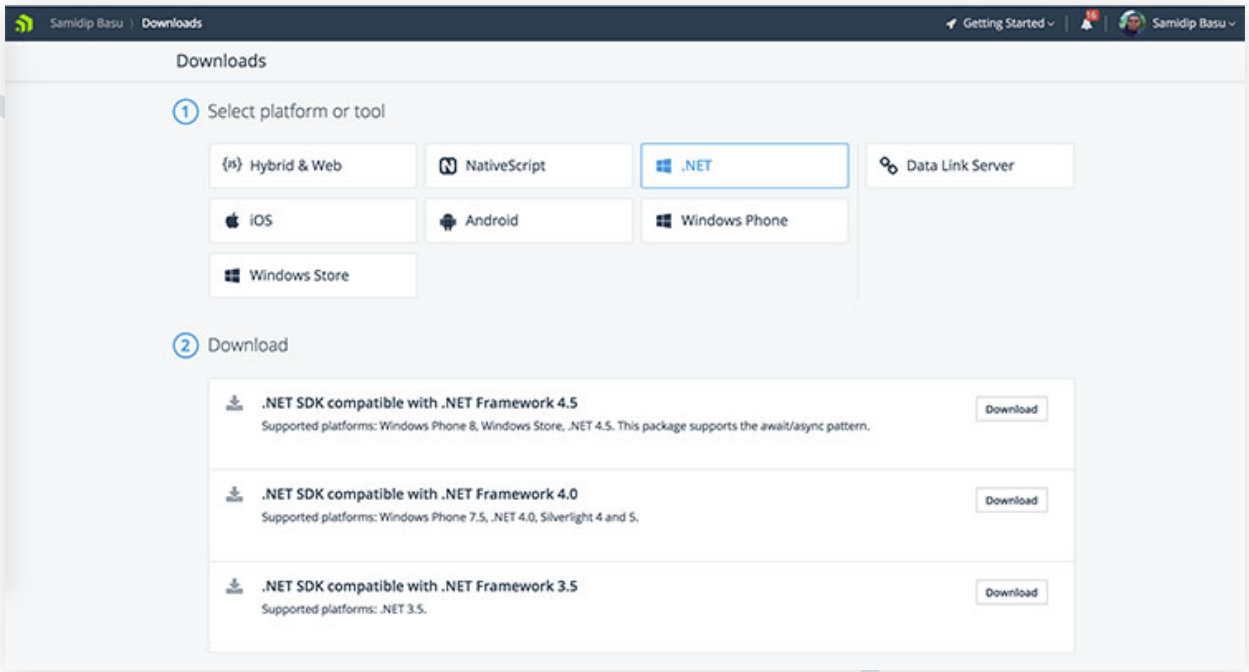
Every content table in Backend Services, like the BizJets one, is essentially an entity—a RESTful endpoint is exposed for each. This endpoint isn't just for reading data, but supports the full range of CRUD operations for easy data manipulation. And every Telerik Platform app gets a unique App ID for basic security. So one could go to https://api.everlive.com/v1/"app-id"/"entityname" in chosen browser and get some JSON back, straight out of the BizJets entity in this case.


```
▼<Response>
  <Count>9</Count>
 ▼<Result>
  ▼<result>
     <AircraftName>Boeing Business Jet</AircraftName>
     <AircraftCapacity>20-25</AircraftCapacity>
    ▼<AircraftImageURL>
       https://bs3.cdn.telerik.com/v1/lmplloffsq4t2dgt/59fcedc0-b071-11e6-8748-ef6880f49a5c
     </AircraftImageURL>
     <CreatedAt>2016-11-22T05:17:22.090Z</CreatedAt>
     <ModifiedAt>2016-11-22T05:17:22.090Z</ModifiedAt>
     <CreatedBy>00000000-0000-0000-0000-000000000000</CreatedBy>
     <ModifiedBy>00000000-0000-0000-0000-000000000000</ModifiedBy>
     <Owner>00000000-0000-0000-0000-000000000000</Owner>
     <Id>f265d8a0-b072-11e6-9683-4f3a0a880ceb</Id>
    ▼<Meta>
     ▼<Permissions>
        <CanRead>true</CanRead>
        <CanUpdate>true</CanUpdate>
        <CanDelete>true</CanDelete>
      </Permissions>
     </Meta>
   </result>
  ▶<result>...</result>
  ▶<result>...</result>
  ▶<result>...</result>
  ▶<result>...</result>
  ▶<result>...</result>
  ▶<result>...</result>
  ▶<result>...</result>
 </Result>
</Response>
```

3. Use Models, ViewModels and Observables

Once you have your cloud data strategy figured out, the next task is how to consume the data in your Xamarin app. Sure you have a RESTful endpoint and all you need is HTTP to communicate with our cloud service. But managing HTTP requests/responses is no fun. What you need is wrappers over the RESTful service. Thankfully, most cloud providers will happily oblige.

Telerik Backend Services provides wrappers for just about any platform—they make it easier to work with cloud data by sugar-coating the underlying HTTP calls. In the Downloads section of the Platform app are the Backend Services SDKs—you'll want to grab the .NET one for Xamarin projects.



Once downloaded, you simply add a reference to the Everlive DLL in your chosen Xamarin project. The code needed to bring cloud data down is pretty generic and not specific to iOS/Android/Windows – so the best place to add this would be the shared PCL or a .NET Standards library.

The easiest way to work with Backend Services data is to have a Model that mimics the entity table in the cloud – the data marshaling comes free. You can craft your model however you like by naming things differently from the entity table or selectively choosing fields, but you'll want to implement the INotifyPropertyChanged pattern for easy two-way UI data binding with the Model's data.

Here's a sample of what a model could look like for the BizJets entity:

```
using System;
using Telerik.Everlive.Sdk.Core;

using Telerik.Everlive.Sdk.Core.Model.Base;
using Telerik.Everlive.Sdk.Core.Serialization;

namespace Aviation
{
    [ServerType("BizJets")]
    public class BizJets : DataItem
    {
        private string aircraftName;
        public string AircraftName
        {
            get
```

```
        {
            return this.aircraftName;
        }
        set
        {
            this.aircraftName = value;
            this.OnPropertyChanged("AircraftName");
        }
    }

    private string aircraftCapacity;
```

With our Model in place, you can already see the basics of a common way to add structure to your Xamarin.Forms apps—the MVVM pattern. The goal is separation of concerns—the Model interacts with your data source, the View is only for UI and the ViewModel plays broker between the two while supporting user interactions. As we have talked about, there are several popular MVVM frameworks that do the heavy lifting for you but there is nothing stopping you from rolling your own loose MVVM pattern. Once you have your Model lined up, next in line would be the ViewModel to actually hydrate the View with data.

Here's some sample code in a ViewModel that can be used to bring down the list of business jets from the cloud using the Model defined earlier:

```
using System;
using System.Collections.ObjectModel;
using System.Threading.Tasks;

using Telerik.Everlive.Sdk.Core;

namespace Aviation
{
    public class BizJetsViewModel
    {
        private string BSAppId = “your-app-id”;
        private EverliveApp ELHandle;

        public ObservableCollection<BizJets> BizJetsCollection { get; set; }

        public BizJetsViewModel()
        {
            EverliveAppSettings appSettings = new EverliveAppSettings() { AppId = BSAppId,
UseHttps = true };
            ELHandle = new EverliveApp(appSettings);
        }
```

```
        public async Task<ObservableCollection<BizJets>> GetAllBizJets()
        {
            var bizJetsManager = ELHandle.WorkWith().Data<BizJets>();
            var allBizJets = await bizJetsManager.GetAll().ExecuteAsync();
            BizJetsCollection = new ObservableCollection<BizJets>();
            foreach (BizJets serializedBizJet in allBizJets)
            {
                BizJetsCollection.Add(serializedBizJet);
            }

            return BizJetsCollection;
        }
    }
}
```

The important thing to take note of in the code above is the ObservableCollection—a special .NET generic class that helps in easy two-way data binding with UI. It houses the strongly-typed business jets collection object and is used as a placeholder once the Everlive handler pulls down the list from the cloud. Notice the async-await pattern as well—you'll want your cloud data fetches to be seamless and not affecting the app responsiveness. All that cloud data is now sitting local in a C# object ready to be data bound and displayed in the UI.

4. Use a Feature-Rich ListView

As is the case, most apps need to display a list of things. And most often, the developer's choice of user interface control is the ubiquitous ListView. Sure you can start with the vanilla ones and build features of your own, but wouldn't you rather focus on your app's functionality, than sweating over perfecting the ListView's interactivity?

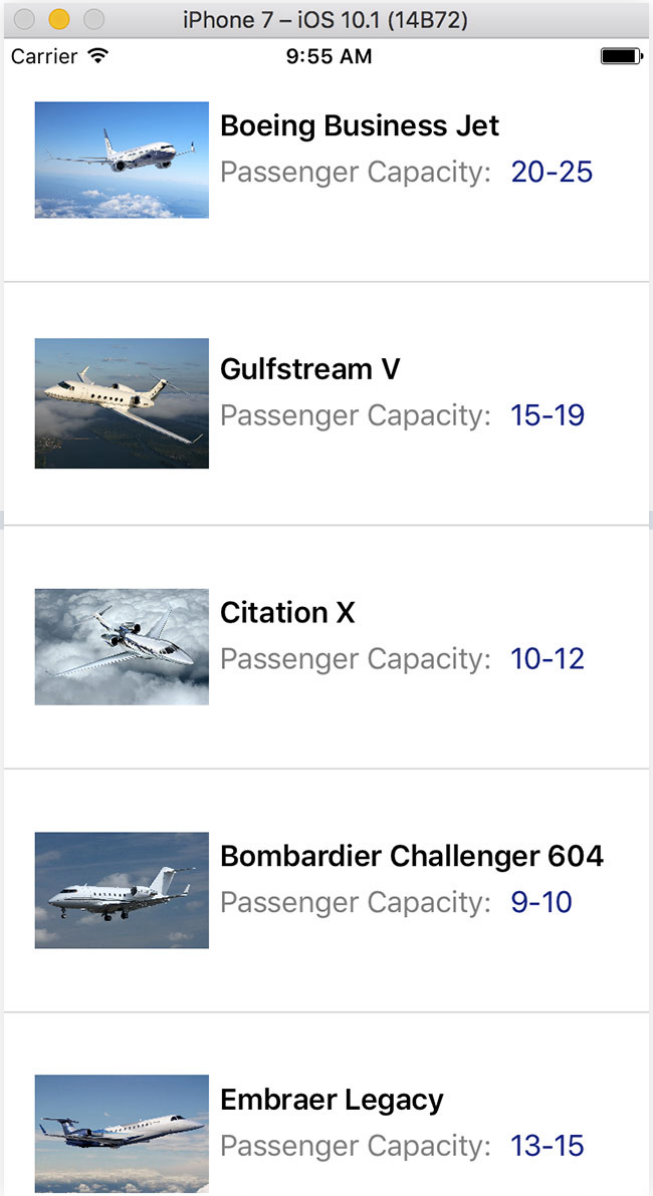
Call us biased, but we think you'll be better served using well-engineered polished and performant UI controls out of the box—like the cross-platform Xamarin.Forms ListView that comes with Telerik UI for Xamarin. First order of business is to set up some XAML to use the Telerik RadListView to display the list of business jets. Here's some simplified markup in the first page (HomeView) of the Aviation app:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.
microsoft.com/winfx/2009/xaml" x:Class="Aviation.BizJetsView"
    xmlns:TelerikDataControls="clr-namespace:Telerik.XamarinForms.
DataControls;assembly=Telerik.XamarinForms.DataControls"
    xmlns:TelerikListView="clr-namespace:Telerik.XamarinForms.DataControls.
ListView;assembly=Telerik.XamarinForms.DataControls">
  <ContentPage.Content>
    <TelerikDataControls:RadListView x:Name="BizJetsList"
SelectionMode="Single">
      <TelerikDataControls:RadListView.ItemTemplate>
        <DataTemplate>
          <TelerikListView:ListViewTemplateCell>
          <TelerikListView:ListViewTemplateCell.View>
            <StackLayout Orientation="Horizontal" Padding="20">
```

```
                        <Image Source="{Binding AircraftImageURL}" WidthRequest="100" HeightRequest="100" />
                        <StackLayout Orientation="Vertical" Padding="0,20,0,0">
                            <Label Text="{Binding AircraftName}" TextColor="Black" FontAttributes="Bold" />
                            <StackLayout Orientation="Horizontal">
                                <Label Text="Passenger Capacity: " TextColor="Gray" />
                                <Label Text="{Binding AircraftCapacity}" TextColor="Navy" />
                            </StackLayout>
                        </StackLayout>
                    </StackLayout>
                </TelerikListView:ListViewTemplateCell.View>
            </TelerikListView:ListViewTemplateCell>
        </DataTemplate>
    </TelerikDataControls:RadListView.ItemTemplate>
</TelerikDataControls:RadListView>
    </ContentPage.Content>
</ContentPage>
```

And finally, the XAML.cs code behind file where things are stitched together. Notice how the corresponding ViewModel is leveraged here—simply fetch the data and data-bind to the ListView.

```
using System;
using System.Collections.Generic;

using Xamarin.Forms;

namespace Aviation
{
    public partial class BizJetsView : ContentPage
    {
        public BizJetsView()
        {
            InitializeComponent();
            this.FetchViewData();
        }

        private async void FetchViewData()
        {
            BizJetsViewModel BZVM = new BizJetsViewModel();
            await BZVM.GetAllBizJets();
            this.BizJetsList.ItemsSource = BZVM.BizJetsCollection;
        }

    }
}
```



Time to run the app? Sure, you get the simple list of business jets with the Telerik ListView bound to cloud data that was pulled down. Only the iOS Simulator is shown here for brevity—Android and UWP behave the same way.

The simplistic looking ListView showing the collection of business jets doesn't do justice to what's under the covers—a ListView on steroids. Users of modern mobile apps expect some gestures and features to be standard on ListViews. Here are some benefits that the Telerik ListView gives you ready out of the box:

- Extreme fine-tuned performance
- Fluid data virtualization for managing data binding to big datasets
- Pull to refresh gesture support
- Customizable Cell swipe gestures
- Sorting, Filtering and Grouping
- Custom styling and intuitive row templates
- Load on demand for incremental data loads
- Row reorder gesture support

Wouldn't you rather focus on your app's functionality than stress over supporting all these ListView features by hand? Here's how to turn on Pull to Refresh on the Telerik ListView—simply implement the RefreshRequested() public event in the code-behind which is triggered by the pull-to-refresh gesture:

```
<TelerikDataControls:RadListView x:Name="BizJetsList" SelectionMode="Single" ItemTapped="BizJet_Selected"
                                 IsPullToRefreshEnabled="true" RefreshRequested="BizJetsList_RefreshRequested">
        <TelerikDataControls:RadListView.ItemTemplate>
            ...
            ...
        </TelerikDataControls:RadListView.ItemTemplate>
</TelerikDataControls:RadListView>
```

Here's how to turn on automatic Load on Demand as the user is scrolling on your rendered ListView. Once again, you get a simple event to respond to:

```
<TelerikDataControls:RadListView x:Name="BizJetsList" SelectionMode="Single" ItemTapped="BizJet_Selected"
                                 IsLoadOnDemandEnabled="true" LoadOnDemandMode="Automatic"
LoadOnDemand="BizJetsList_LoadOnDemand">
        <TelerikDataControls:RadListView.ItemTemplate>
            ...
            ...
        </TelerikDataControls:RadListView.ItemTemplate>
</TelerikDataControls:RadListView>
```

5. Decide on Local Data Persistence

As soon as you start building out your Xamarin.Forms app, you'll have to figure out a consistent data persistence strategy—how do you actually store data on the device from your app? Sure you could pull down data from the cloud every time, but then your app will completely stop working when there is no connectivity.

Ideally you want to cache data once it has been pulled down, so your app keeps working graciously without connectivity. Additionally, most apps require some user and app specific data to be persisted between multiple uses of the app.

There are various data persistence strategies when it comes to Xamarin.Forms apps. What you ideally want is a cross-platform model that works everywhere and allows you data management from your shared code. Sure there are SQLLitePCL solutions and Settings Plugins, but the easiest option may be a built-in dictionary for cross-platform data persistence.

All Xamarin.Forms apps use the Application class to bootstrap themselves, and in there is a Properties dictionary. This dictionary can be used to store serialized primitive data in key value pairs. The Properties dictionary is saved to the device automatically and you can access this data from anywhere in your Xamarin.Forms code using Application.Current.Properties.

Here's some code from the ViewModel that pulls in a list of business jets for the corresponding View. Notice the GetAllBizJets() method—it first checks if the data is already cached, and if so, pulls it out of the Properties dictionary. If not, the code goes off to the cloud, pull down data and stick the whole collection of objects into the Properties dictionary for app-wide persistence.

```
public class BizJetsViewModel
       {
            private string BSAppId = "<Your Backend Services App Key>";
            private EverliveApp ELHandle;

            public ObservableCollection<BizJets> BizJetsCollection { get; set; }

            public BizJetsViewModel()
            {
                EverliveAppSettings appSettings = new EverliveAppSettings() { AppId =
BSAppId, UseHttps = true };
                ELHandle = new EverliveApp(appSettings);
            }

            public async Task<ObservableCollection<BizJets>> GetAllBizJets()
            {
                BizJetsCollection = new ObservableCollection<BizJets>();
```

```
            if (Application.Current.Properties.ContainsKey("BizJetsCollection"))
            {
                BizJetsCollection = Application.Current.Properties["BizJetsCollection"] as
ObservableCollection<BizJets>;
                return BizJetsCollection;
            }
            else
            {
                var bizJetsManager = ELHandle.WorkWith().Data<BizJets>();
                var allBizJets = await bizJetsManager.GetAll().ExecuteAsync();

                foreach (BizJets serializedBizJet in allBizJets)
                {
                    BizJetsCollection.Add(serializedBizJet);
                }

                Application.Current.Properties["BizJetsCollection"] = BizJetsCollection;
                return BizJetsCollection;
            }
        }
    }
```

6. Navigate With Ease

Almost every mobile app is a collection of pages—navigation through the pages is what makes up the user experience. Xamarin.Forms apps are no exception and, thankfully, an easy navigation model is built in. Developers stitch together a stack of pages that follows a Last-in-First-out (LIFO) pattern. Navigation from one page to another pushes a page into the stack and returning to the previous page pops it out of the stack. Simple.

All you have to do to leverage the built-in Xamarin.Forms navigation model is to use the NavigationPage class, which manages the stack automatically for you. Using the NavigationPage class adds a navigation bar to the top of the page, complete with title and icons if desired. Users also see a back navigation button as they navigate—this is customized for each mobile platform.

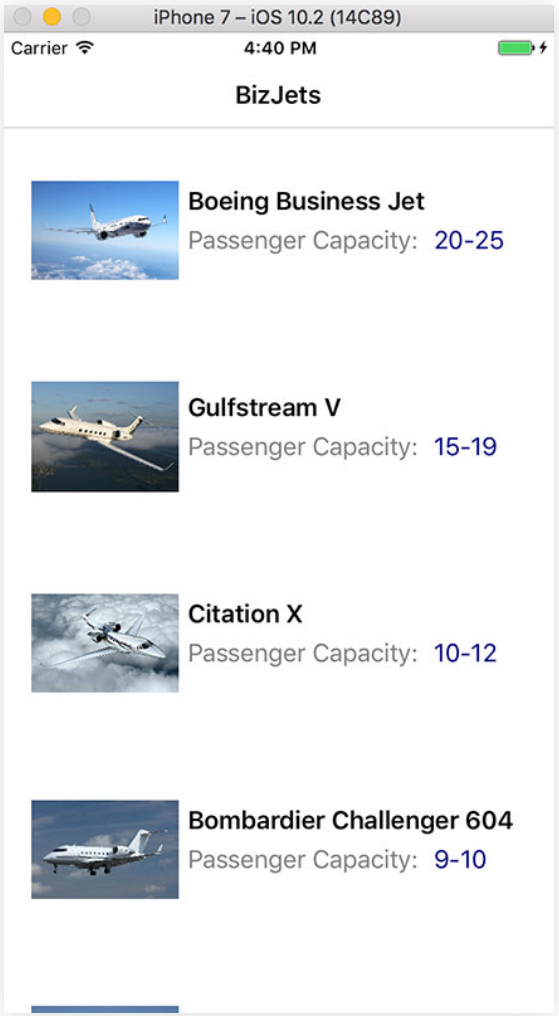
You'll preferably want to use the NavigationPage class from the very first page of your app, which is defined in the AppStart.cs in your shared PCL code. Here's how:

```
public class App : Application
{
public App()
{
        MainPage = new NavigationPage(new HomeView());
}
    ..
}
```

The first view (HomeView) of our Aviation app is already hooked up to the NavigationPage class. We have the Telerik ListView display our list of business jets, letting us allow single selections and navigate the user to a detail page. This is the ubiquitous master-detail scenario and we would like to carry the context of the selected business jet onto the detail page. Here's some code:

```
        <TelerikDataControls:RadListView x:Name="BizJetsList" SelectionMode="Single" ItemTapped="BizJet_
Selected"
        ...
                ...
        </TelerikDataControls:RadListView>


        // Code behind.
        private async void BizJet_Selected(object sender, Telerik.XamarinForms.DataControls.ListView.
ItemTapEventArgs e)
        {
                BizJets selectedBizJet = (BizJets)e.Item;
                await Navigation.PushAsync(new BizJetDetailView(selectedBizJet));
        }
```

Notice the use of PushAsync() method – push the detail page into the navigation stack, carry the selected business jet and take the user to the detail page. As a free perk, the navigation model renders the appropriate Back button, which brings the user back to the ListView page. Here's the simple navigation in our Aviation app:



If you every wanted to handle the back navigation programmatically, simply use the PopAsync() method to remove the navigated page out of the stack, like so:

```
await Navigation.PopAsync();
```

7. Organize Content

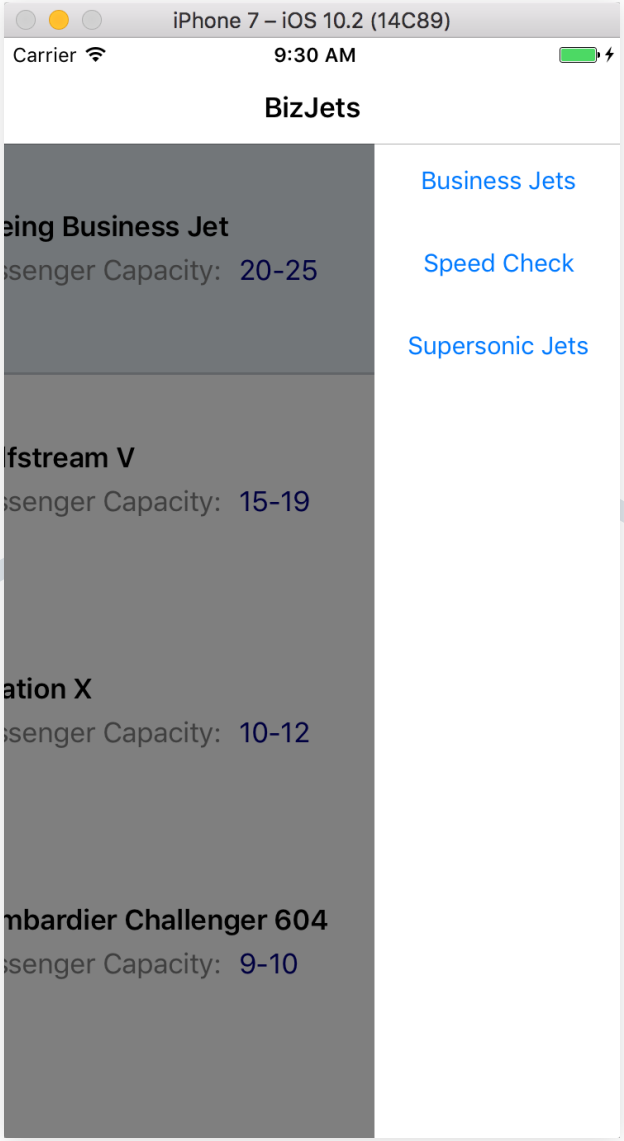
One of the keys to a successful mobile app is content organization. Pages that make up the app should be easily discoverable and a snap to navigate in between. One common technique is to tuck away content that the user can discover with a gesture and allow navigation to other parts of the app.

The SideDrawer in Telerik UI for Xamarin is a master at this trick—it hides away an entire view behind a swipe gesture that the user can discover easily. The SideDrawer is simply a placeholder commonly used to house navigation, app/user settings or other appropriate UI. A little flick gesture from any configured edge of the screen allows the user to visualize the SideDrawer content, often accompanied by open/close transition animations. Here's a simple SideDrawer in the Aviation app's home page:

```
<TelerikPrimitives:RadSideDrawer x:Name="HomeDrawer" DrawerLength="150" DrawerLocation="Right">
        <TelerikPrimitives:RadSideDrawer.MainContent>
                <TelerikDataControls:RadListView x:Name="BizJetsList"
                ...
                ...
                </TelerikDataControls:RadListView>
        </TelerikPrimitives:RadSideDrawer.MainContent>
        <TelerikPrimitives:RadSideDrawer.DrawerContent>
                 <StackLayout>
                     <Button Text="Business Jets" Clicked="BizJet_Clicked" />
```

```
                                <Button Text=”Speed Check”
Clicked=”SpeedCheck_Clicked” />
                            <Button Text=”Supersonic Jets”
Clicked=”SupersonicJet_Clicked”/>
                        </StackLayout>
                </TelerikPrimitives:RadSideDrawer.DrawerContent>
        </TelerikPrimitives:RadSideDrawer>
```

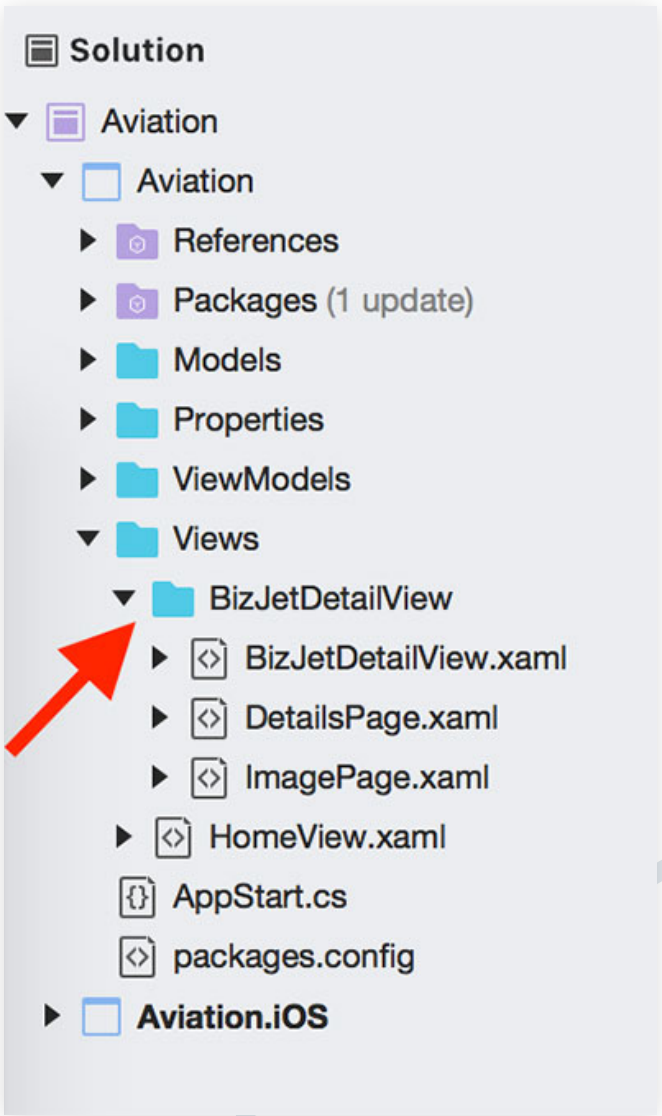
As you see, the SideDrawer simply presents a placeholder to house hidden content—in this case, three simple navigation buttons towards other parts of the app, as seen on the right. And yes, you can render hamburger icons to provide a subtle hint to guide the user to the side drawer. Go organize content to your heart's content.



8. Present Content

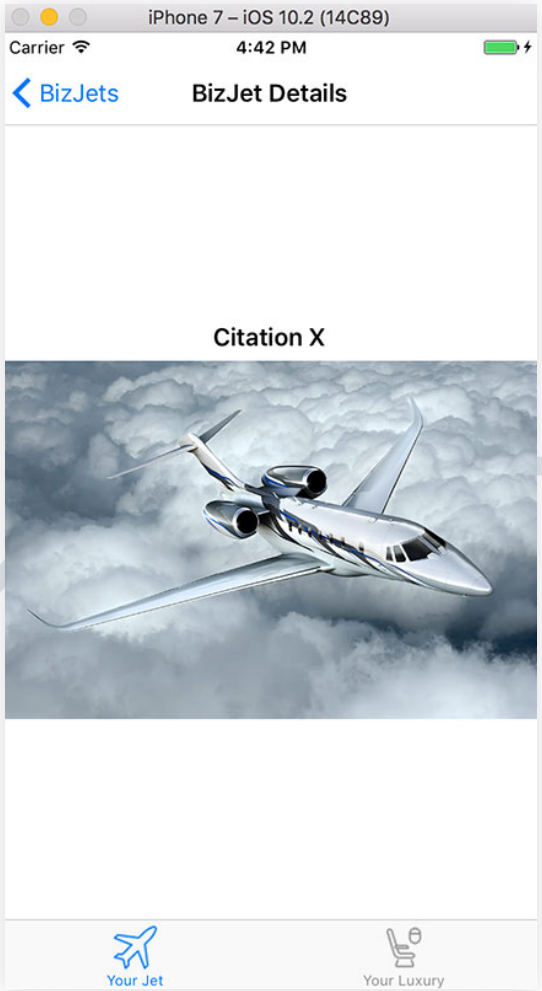
While there are many ways to present content in your apps, one of the most common paradigms is Tabs because the user intuitively knows how to use them. With Xamarin.Forms, you have a built-in way to present app content as tabs through the TabbedPages class. The user interface presents a list of tabs customized to each platform and a large detail area. Here's some simple code in the Detail page of the Aviation app to display business jet information:

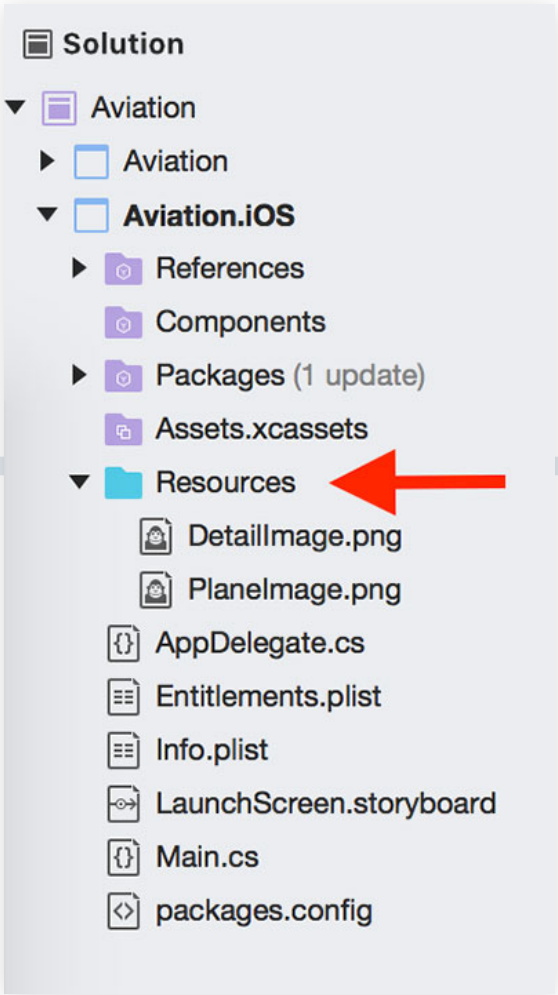
```
<?xml version="1.0" encoding="UTF-8"?>
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Aviation.BizJetDetailView"
    xmlns:MyTabPages="clr-namespace:Aviation;assembly=Aviation">
    <TabbedPage.Children>
        <MyTabPages:ImagePage Title="Your Jet" Icon="PlaneImage" />
        <MyTabPages:DetailsPage Title="Your Luxury" Icon="DetailImage" />
    </TabbedPage.Children>
</TabbedPage>
```



Notice how we are trying to fill in the Children collection of a TabbedPage. These are individual completely customizable ContentPages, just tied together as a list of Tabs. It may be a good idea to keep the children pages that make up the Tabs in one place—like in a folder.

Also, for iOS apps, simply setting the Icon property on the tabs adds the image icons. Just make sure to have appropriately sized icons in the Resources folder of your iOS project. The colored state of the icons also comes free.





One of the challenges you may face with tabbed pages is data management—each of the pages is a separate entity, but they together make up a view and often share data. One common way to solve this is by using a unified ViewModel and a single instance of it.

```
public class BizJetsDetailViewModel
{
    private static readonly BizJetsDetailViewModel SingletonBJDVM = new BizJetsDetailViewModel();
    private BizJets IndividualBizJet;

    private BizJetsDetailViewModel()
    {
        // No public constructor.
    }

    public static void SetBJDViewModel(BizJets SelectedBizJet)
    {
        SingletonBJDVM.IndividualBizJet = SelectedBizJet;
    }

    public static BizJets GetIndividualBizJet()
    {
        return SingletonBJDVM.IndividualBizJet;
    }
}
```

Notice how the ViewModel exposes methods to set/get access to the selected BizJet – and this is shared among all children of TabbedPage through the singleton instance. Now the parent detail view may set the selected BizJet as received through navigation context and all children pages get access to it.

```
public BizJetDetailView(BizJets SelectedBizJet)
{
    BizJetsDetailViewModel.SetBJDViewModel(SelectedBizJet);
    InitializeComponent();
}
```

9. Manage the App Lifecycle

Every mobile app goes through several life cycle phases as intermittent usage continues through the day. These change of states offer a great opportunity for the app developer to take appropriate action to ensure the best possible user experience. Xamarin.Forms makes this quite easy with events that are raised as the app changes state—the developer gets to respond in the appropriate event handler.

The Application class, which is where all Xamarin.Forms apps begin life, exposes three virtual methods that can be overridden to handle lifecycle events:

1. OnStart – Triggered when the application starts up.

2. OnSleep – Triggered each time the application goes to the background.

3. OnResume – Triggered when the application is resumed, after being sent to the background.

Look into the AppStart.cs file and you'll find hooks for each of the lifecycle events—perfect opportunities for you to stow away user/app state and preserve user experience continuity.

```
public class App : Application
{
    ...

    protected override void OnStart()
    {
    }

    protected override void OnSleep()
    {
    }

    protected override void OnResume()
    {
    }
}
```

Remember the Properties dictionary we talked about for data persistence? The Properties collection is saved automatically to disk on each mobile platform during the OnSleep() event. There is however, a new SavePropertiesAsync() method, which can be called to proactively persist the Properties dictionary at any time, in case the app crashes or is killed by the OS. There is no trigger as to when your app terminates, so the best practice may be to preserve all your data and state information during OnSleep() and retrieve them back during OnResume().

10. Be Smart With Images

Imagery delights users and carefully used visuals often leads to a great user experience. For Xamarin. Forms app developers, handling images is quite easy—the perfect opportunity to brand your app correctly and keep inviting users back with rich imagery. You may display local images that you ship with your app package or pull them down over the wire.

The Image control in Xamarin.Forms displays images with full customization. Images are often displayed as a part of a page or inside a container, like in the case of the detail page for business jets:

```
<ContentPage.Content>
    <StackLayout Orientation=”Vertical” VerticalOptions=”Center”>
        <Label x:Name=”BizJetName” TextColor=”Black” FontAttributes=”Bold”
HorizontalOptions=”Center” />
        <Image x:Name=”BizJetImage” Aspect=”AspectFit” />
    </StackLayout>
</ContentPage.Content>
```

One important property of the Image control is the Aspect—this determines how the image is scaled to occupy the display area. The options are:

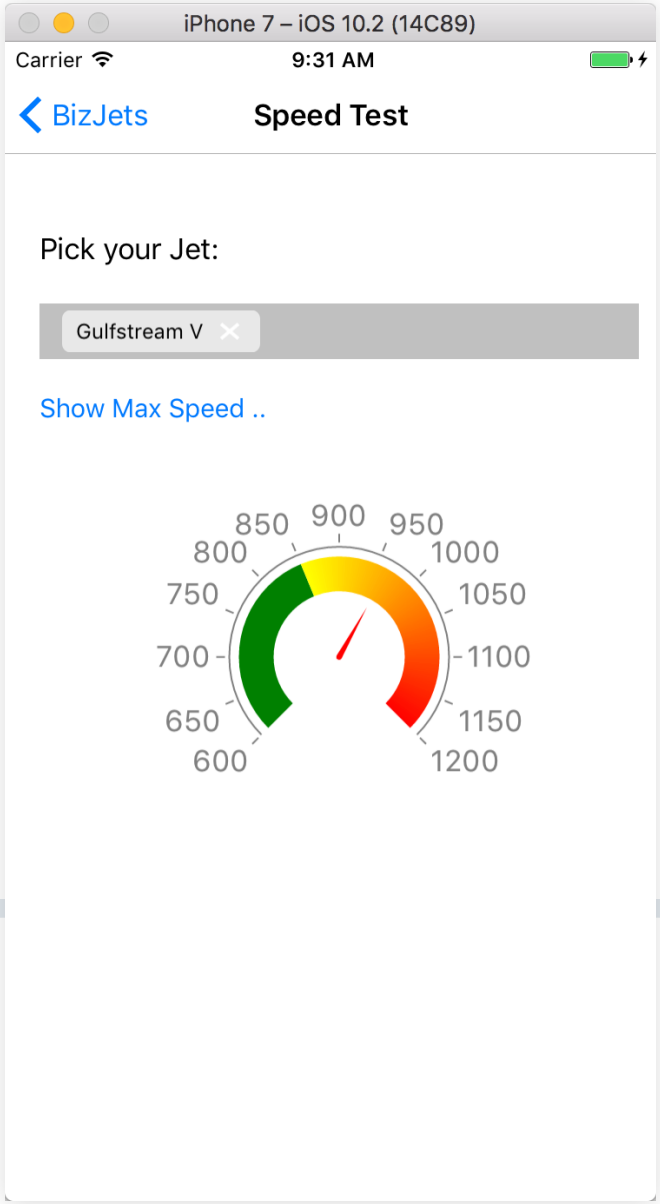
- AspectFill – Scale image to fill the view. Some image parts may be clipped, but no distortion happens.
- AspectFit – Scale image to fit the view. Some parts may be left empty (letterboxing).
- Fill – Scale image to exactly fill the view. Scaling may not be uniform in X & Y, and may cause image distortion.

Downloading imagery is great way to keep your app content fresh, but you'll want to ensure that you are not misusing the user's data bandwidth. The solution is to cache images once downloaded – and reuse on subsequent views.

Thankfully, caching functionality is built-in with the ImageSource instance from where the Image control downloads/displays the image, unless you turn it off explicitly. You may also control the duration after which the cached images become stale.

Here's some code to display the business jet images – notice how the Source property is data bound and the cache is being set for 5 days. You get to adjust image caching to your specific app needs.

```
public ImagePage()
{
    InitializeComponent();

    BizJets IndividualBizJet = BizJetsDetailViewModel.GetIndividualBizJet();

    BizJetName.Text = IndividualBizJet.AircraftName;
    BizJetImage.Source = new UriImageSource
    {
        Uri = new Uri(IndividualBizJet.AircraftImageURL),
        CachingEnabled = true,
        CacheValidity = new TimeSpan(5, 0, 0, 0)
    };
}
```



11. Use Rich Data Visualizations

Data is often beautiful when visualized—it entices the user to use your app more. However, data visualization UI is also the hardest to create from scratch - performance, cross platform compatibility and the sheer complexity of the UI are daunting. This is also where Telerik UI for Xamarin can help with its rich suite of data visualization UI out of the box.

Say you want to display the max speed of some of the business jets for customers—essentially, plotting a value over a range between minimum and maximum. The Radial Gauge does a great job of this. Here's a sample UI:

The radial gauge allows for complete customization of the axis showing range of values as well as the needle indicator. Here's the code powering the gauge UI on the left:

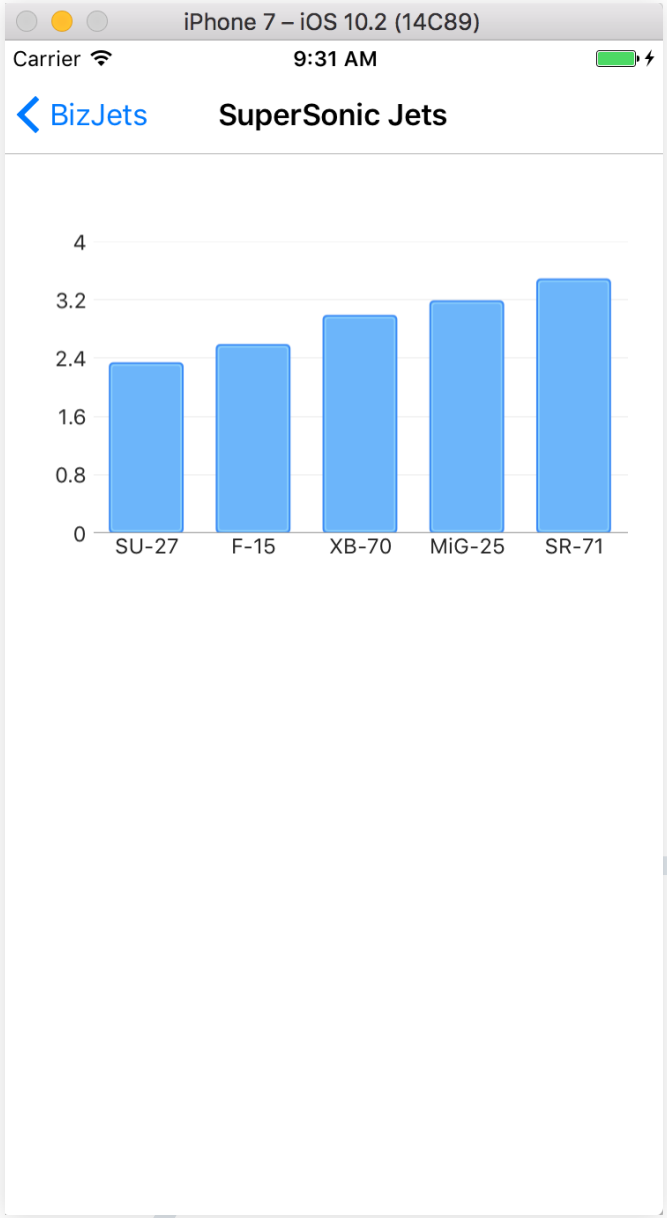
```
        <TelerikGauges:RadRadialGauge Margin=”0,25,0,0”>
                  <TelerikGauges:RadRadialGauge.Axis>
                    <TelerikGauges:GaugeLinearAxis Minimum=”600” Maximum=”1200” Step=”50” />
                  </TelerikGauges:RadRadialGauge.Axis>
                  <TelerikGauges:RadRadialGauge.Indicators>
                    <TelerikGauges:GaugeNeedleIndicator x:Name=”SpeedIndicator” Value=”900”
Offset=”30” Position=”Start” />
                    <TelerikGauges:GaugeShapeIndicator Value=”80” />
                  </TelerikGauges:RadRadialGauge.Indicators>
                  <TelerikGauges:RadRadialGauge.Ranges>
                    <TelerikGauges:GaugeRangesDefinition>
                      <TelerikGauges:GaugeRange From=”600” To=”850” Color=”Green” />
                      <TelerikGauges:GaugeGradientRange From=”850” To=”1200”>
                        <TelerikCommon:RadGradientStop Color=”Yellow” Offset=”850” />
                        <TelerikCommon:RadGradientStop Color=”Red” Offset=”1200” />
                      </TelerikGauges:GaugeGradientRange>
                    </TelerikGauges:GaugeRangesDefinition>
                  </TelerikGauges:RadRadialGauge.Ranges>
        </TelerikGauges:RadRadialGauge>

        // Code Behind:
        this.SpeedIndicator.Value = IndividualJet.JetMaxSpeed;
```

Want fancy charts and graphs for data visualization? You got it—Telerik UI for Xamarin has about a dozen different types of Charts built in. These are feature-rich, intuitive and interactive complex UI controls that are also easy for the developer to implement. Bar charts, Cartesian graghs, Pie charts, Line charts - you get them all with touch-friendly interactions, customizable axes and sharp performance/rendering, all on top of consistent API for developers.

Here's a simplistic Bar chart showing some of the fastest Supersonic jets and their corresponding Mach speeds. (Don't get your hopes too high, none of them are civilian jets):

Here's the markup powering the chart on the right - bound to an Observable Collection of Supersonic Jets with aircraft names and speeds:

```
<TelerikChart:RadCartesianChart>
  <TelerikChart:RadCartesianChart.HorizontalAxis>
    <TelerikChart:CategoricalAxis />
  </TelerikChart:RadCartesianChart.HorizontalAxis>
  <TelerikChart:RadCartesianChart.VerticalAxis>
    <TelerikChart:NumericalAxis />
  </TelerikChart:RadCartesianChart.VerticalAxis>
  <TelerikChart:RadCartesianChart.Series>
    <TelerikChart:BarSeries x:Name="JetDataSeries"
                            CategoryBinding="JetName"
ValueBinding="JetMaxSpeed" />
  </TelerikChart:RadCartesianChart.Series>
</TelerikChart:RadCartesianChart>
```



```
// Code Behind:
SuperSonicJetsViewModel SSVM = new SuperSonicJetsViewModel();
this.JetDataSeries.ItemsSource = SSVM.JetDataSource;
```

12. Save on data bandwidth

Mobile users are often data capped and your app users will love it if you do more with less - the answer is data compression. Your app is expected to perform fast, secure transactions and feel responsive while doing heavy lifting. Enter a non-UI component in Telerik UI for Xamarin—the useful ZipLibrary. Here are some immediate benefits of using the Telerik ZipLibrary in your Xamarin.Forms apps:

- Easy loading/creation of Zip files
- Support for large files, like images and documents
- Consistent .NET API
- Easy transmission of bundled files in one go
- Encryption support with password protection

With a single reference DLL, here's some sample code you could write to open a Zip archive and read contained files one at a time:

```
using (Stream stream = File.Open("SomeFile.zip", FileMode.Open))
{
    using (ZipArchive archive = new ZipArchive(stream))
    {
        // Iterate through contents of zip file using the ZipArchive.Entries property.
    }
}
```

Here's some code to actually create a Zip archive and put a sample text file inside:

```
using (Stream stream = File.Open("test.zip", FileMode.Create))
{
    using (ZipArchive archive = new ZipArchive(stream, ZipArchiveMode.Create, false, null))
    {
        using (ZipArchiveEntry entry = archive.CreateEntry("SampleFile.txt"))
        {
            StreamWriter writer = new StreamWriter(entry.Open());
            writer.WriteLine("Hello world!");
            writer.Flush();
        }
    }
}
```

Conclusion

There will be numerous roadblocks as you build your next cross-platform mobile app—Xamarin.Forms helps you out in many ways. Just make sure to bookmark the Docs and have a consistent strategy on the big ticket items—data management, navigation and app organization. Your goal is to ensure a smooth user experience that keeps inviting the user back to your app, over and over again. And stop reinventing the wheel on complex UI wherever possible. Telerik UI for Xamarin is here to help with rich and performant controls for all your app needs. Keep coding and good luck!

Brought to You by Telerik UI for Xamarin

Telerik® UI for Xamarin by Progress is a collection of UI controls and functionalities that complement the default controls found in the platform. The controls offer many non-trivial scenarios out of the box, helping developers to implement polished UI with native-quality performance in their apps faster, shortening time to market. Telerik UI for Xamarin includes Xamarin.iOS wrappers, Xamarin.Android wrappers and Xamarin.Forms controls.

Telerik UI for Xamarin uses the Xamarin.Forms technology so developers can build native iOS, Android and Universal Windows Platform apps from a single shared C# code base. The Telerik controls for Xamarin.Forms enables developers to easily implement various functionalities in their Xamarin.Forms projects and achieve the same scenarios across platforms using a single shared C# code base.

Try UI for Xamarin

About the Author

Sam Basu is a technologist, author, speaker, Microsoft MVP, gadget-lover and Progress Developer Advocate for Telerik products. With a long developer background, he now spends much of his time advocating modern web/mobile/cloud development platforms on Microsoft/Telerik technology stacks. His spare times call for travel, fast cars, cricket and culinary adventures with the family. You can find him on the internet.

Worldwide Headquarters

Progress, 14 Oak Park, Bedford, MA 01730 USA Tel: +1 781 280-4000 Fax: +1 781 280-4095
On the Web at: www.progress.com
Find us on facebook.com/progresssw twitter.com/progresssw youtube.com/progresssw
For regional international office locations and contact information, please go to www.progress.com/worldwide

About Progress

Progress (NASDAQ: PRGS) is a global leader in application development, empowering the digital transformation organizations need to create and sustain engaging user experiences in today's evolving marketplace. With offerings spanning web, mobile and data for on-premise and cloud environments, Progress powers startups and industry titans worldwide, promoting success one customer at a time. Learn about Progress at www.progress.com or 1-781-280-4000.

Rev 2017/03 | 170306-0045

Progress®