Progress® Telerik® UI for Xamarin

The Ultimate Guide To Xamarin.Forms Mobile Development

With Real-World Sample Applications

EBOOK

Table of Contents

Executive Summary / 5

Mobile Development with Xamarin.Forms / 5

- The Promise of Cross-Platform Development / 5
- Xamarin Native vs. Xamarin.Forms: Which Is Best for Your Next Cross-Platform Mobile Application? / 6

Best Practices with Progress® Telerik® Xamarin.Forms Demo Applications / 7

- Telerik ERP: Advanced Enterprise Resource Management App / 7
- Telerik ToDo: Customer-Facing Task Management App / 8
- Telerik ToDo: Customer-Facing Task Management App / 8
- The Tech Behind the Telerik Demo Applications: What We Used and Why / 9
  - Microsoft Azure / 9
  - MVVM Framework / 9

Telerik ERP: How to Build a Mobile Enterprise Resource Management Application with Xamarin.Forms / 10

- The Backend Structure of Telerik ERP / 10
  - Hosting Unstructured Data / 11
  - Hosting Structured Data / 11
- Telerik ERP: MVVM Framework and UIs / 15
  - ERP MvvmCross Implementation / 15
  - Services / 16
  - ViewModels / 17
- Views / 19
- Views UI / 21
  - Xamarin Charts / 21
  - Xamarin DataGrid / 24
  - Xamarin BusyIndicator / 25
  - Advanced Xamarin ListView / 25

Telerik CRM: How to Build a Mobile Customer Relationship Management Application with Xamarin.Forms / 28
Azure App Service Walkthrough / 28
Configuration and Starter Projects / 28
Entities / 30
DbContext / 31
Database Initializer / 32
Table Controllers / 33
Publish to Azure / 35

Building the UI with Telerik UI for Xamarin / 37
Installing Telerik UI for Xamarin / 37
Azure Mobile Service Client SDK / 40
Data Models / 40
Service Classes / 41
Views and ViewModels / 42
EmployeesViewModel / 43
EmployeesPage / 46
RadListView and RadEntry / 46
RadBusyIndicator / 47
EmployeeDetailViewModel / 49
EmployeeDetailsPage / 51
EmployeeEditViewModel / 55
EmployeeEditPage / 56
Wrapping Up / 60

Powering the Conversation / 60
Intents / 61
Training / 62
Testing / 62
Publishing / 63
API Endpoints / 64

Azure Bot Service / 65
First Use: Testing with WebChat / 66
Explore the Code / 66

Xamarin.Forms Chat Service / 69

Bot Service Class and DirectLine API / 69

ViewModel / 72

The XAML / 74

Typing Indicators / 74

Message Styling / 74

Connecting the Bot to LUIS / 75

Telerik ToDo: How to Build a Mobile Task Management Application with Xamarin.Forms / 77

Overview of the Technologies and Frameworks / 77

Application Structure / 78

MVVMFresh: A Lightweight MVVM Framework Implementation for Xamarin.Forms / 79

Data Access / 79

Services / 80

Frontend Implementation: UI Layer and Telerik UI for Xamarin Controls / 82

Conclusion / 89

Executive Summary

Cross-platform mobile development is a great way for developers to reuse their existing programming skills and deploy fully-functional mobile applications on multiple platforms from a single shared codebase. It also presents an opportunity for decision-makers looking to reduce the overhead of their mobile development investment.

In this whitepaper, we will walk you through three real-world Xamarin.Forms application examples, their backend architecture and data access, UI layer and source code. If you are a .NET developer looking to expand your understanding of Xamarin.Forms application development by using real sample apps built by fellow developers—you have come to the right place!

Mobile Development With Xamarin.Forms

The Promise of Cross-Platform Development

As access to business-critical data has become a must-have prerequisite for the day-to-day operations of most companies, mobile devices are now an imperative part of today's business landscape. As the types of devices on which software applications run proliferate, businesses are faced with the challenge of ensuring their applications run smoothly on any device and platform, which brings a massive overhead in their mobile development investment.

From a development standpoint, modern native mobile development can be too limited and domain-dependent for software engineers who have experience with various stacks, especially those focused on developing line-of-business applications. Knowing Java and Objective-C is great if you are serious about developing complex and GUI-heavy apps for Android and iOS, but if you happen to be a .NET developer working on enterprise applications, your options for developing mobile apps are rather limited.

Fortunately, cross-platform development for .NET came to the rescue in the face of Xamarin. Developers around the world cheered the possibility of deploying their mobile applications on multiple devices from a single C# codebase. As with most technological innovations, reality proved more complex than the initial hype.

While Xamarin is currently the most stable and mature cross-platform framework on the market designed to cater to the .NET developer community, it comes with its own set of challenges, especially for people who haven't had much experience in mobile development but want or need to reuse their .NET expertise in that field.

Don't get us wrong, the Xamarin framework does deliver on its promise—it provides developers with a multifaceted toolset for building iOS and Android applications using C# as a cross-platform language and .NET as the base framework for both platforms. Yet, before you dive into your Xamarin project, do take several things into consideration.

Xamarin Native vs. Xamarin.Forms: Which is Best for Your Next Cross-Platform Mobile Application?

When setting the objectives of your mobile development project, first you need to decide whether to take the classical, Xamarin Native approach to development or use Xamarin.Forms. Both approaches allow you to share your data access layer and business logic, but they have their pros and cons you should be mindful of.

If you go down the Xamarin Native road, you can reuse any and every available API for creating a truly platform-specific, native experience for the user. This will entail doing some native iOS or Android tinkering, especially when it comes to updates and application maintenance.

With Xamarin.Forms on the other hand, you define a single UI layer that is reused across all platforms. On the plus side, updates and maintenance are significantly simplified since you'd only need to update a single source file.

If your application requires platform-specific experience, you should consider Xamarin Native. If you are looking for ultimate code reuse, faster time to market and having a unified application UI and behavior across all platforms, Xamarin.Forms is your implementation of choice.

Best Programming Practices with Progress® Telerik® UI for Xamarin Demo Applications

The ancient Greeks had two words that conceptualized wisdom—one was *sophia*, the wisdom you gain from theory, and the other *phronesis*, the wisdom you gain from practical experience. The truth is, all the theory in the world won't help you build anything of value until you tinker with it in a more practical manner.

Phronesis is considered the virtue of the true inventors and one of the best ways of learning—by practice and example. We would like to provide you with the practical knowledge of building powerful Xamarin.Forms applications. As developers, we have skin in this game and would like to give you our best tips about developing your next great Xamarin.Forms app.

In the next sections, we will look at the applications showcased in this whitepaper and make a quick overview of the backend and frontend technologies we used to develop our Progress Telerik UI for Xamarin sample applications.

Telerik ERP: Advanced Enterprise Resource Management App

When we embarked on the journey of creating sample Xamarin.Forms applications with our own Progress Telerik UI suite, we wanted to deliver several real-life, feature-rich and functional applications that cover the more common LoB app scenarios. Telerik ERP is one of them.

The Telerik ERP app is an enterprise resource management application built from the ground up with Xamarin.Forms, Microsoft Azure Services, the MVVM framework and Progress Telerik UI for Xamarin.

The application enables its users to quickly and easily access and manage business-critical information regarding relations with customers and vendors, business transactions and the latest updates and follow-ups on products and orders. In todays' complex and dynamic global economy, the Telerik ERP app is a must-have for any business that requires remote access and monitoring of their business information. The Telerik ERP sample app masterfully combines Microsoft Azure Services to host its data in the cloud, the MVVM framework to maximize the shared code across Android, iOS and UWP devices and Telerik UI for Xamarin frontend to achieve a fluid user experience.

Telerik CRM: Customer Relations Management App

Telerik CRM is another great example of a fully functional, cross-platform Xamarin.Forms demo application. The app was developed for a functioning art gallery to manage its customer relations. It's built using a combination of Azure App Services for the backend, Azure Bot Framework for an in-app support chat service and Telerik UI for Xamarin for a beautiful and feature-rich client experience.

The application provides a seamless user experience that enables users to easily explore information about employees, customers, products and orders. It allows customers to interact with a support chat bot that will guide them through the interface and answer various questions about finding appropriate information.

The Telerik CRM application utilizes three technologies: ASP.NET backend hosted with Azure App Services, using SQL Server for its database, Azure Bot Framework, part of Microsoft Cognitive Services suite, using a custom LUIS (Language Understanding) for an AI-power, machine-learning trained in-app bot support service and—Telerik UI for Xamarin suite powering the frontend of the application.

Telerik ToDo: Customer-Facing Task Management App

If your next project is building a customer-facing app and you are wondering where to begin, look no further—Telerik ToDo is a simple, yet elegant customer-facing application designed to cover various task management scenarios. The application's frontend is built entirely with Telerik UI for Xamarin controls, while the backend is powered by a variety of interesting technologies, such as Entity Framework Core, SQLite and MMVMFresh.

The Telerik ToDo sample application is a perfect example of a lightweight popular app that can be built with Xamarin.Forms in no time. The application's data access capability is powered by Entity Framework Core — an open-source, cross-platform version of the popular Entity Framework. The database engine is built with SQLite and the structural design with MVVMFresh.

The Tech Behind the Telerik Demo Applications: What We Used and Why

Before we dig deeper into the development of our Xamarin.Forms demo applications, we would like to outline some of the technologies we used in addition to our own Telerik UI for Xamarin library, which powers the beautiful frontend of the apps, and why you should consider using them too.

Microsoft Azure

Throughout the development process of our mobile apps, we heavily relied on the rich variety of Microsoft Azure suite. From data hosting in the cloud to creating an AI-powered chatbot, our sample apps leverage the capabilities of Microsoft Azure. Here are some of the technologies we used:

- Microsoft Azure Services for hosting the data of the Telerik ERP application in the cloud
- Azure App Services for hosting the ASP.NET backend and SQL database of the Telerik CRM application
- Azure Bot Framework for the AI-power, machine-learning trained in-app bot support service of the Telerik CRM application

MVVM Framework

Every single Telerik demo application has been built with an MVVM framework implementation. There are multiple benefits of using the MVVM framework for Xamarin.Forms development. The framework provides you with all the tools you need to create a great Xamarin.Forms app, including dependency injection, viewmodel-view association, navigation, messaging and more. The framework itself has multiple implementations and we have used a couple in our app development:

- MvvmCross: Built for the Xamarin development community. We used MvvmCross for our ERP sample application

- MVVMFresh: A super lightweight MVVM framework created specifically for Xamarin.Forms and designed to be easy, simple and flexible to use

We covered most of what is invisible to the user, backend and architecture-wise. But the most crucial element of an application, its interaction point with users, is still to be discussed. We built our Telerik Xamarin.Forms sample applications using only controls from our own Telerik UI for Xamarin suite. ListView, TreeView, Popups, conversational UI, DataGrid—our UI library has every component you will ever need to build truly stunning Xamarin.Forms applications.

Now, let's see how all these technologies tie together to create a few real-world Xamarin. Forms applications.

Telerik ERP: How To Build A Mobile Enterprise Rersource Management Application With Xamarin.Forms

In this section, we will review the backend and frontend structures of our Enterprise Resource Planning (Telerik ERP) demo application and the bundle of technologies we used in the development process.

The Backend Structure of Telerik ERP

The backend of the Telerik ERP application is hosted on Microsoft Azure Mobile Apps, which allows you to quickly cloud-enable your mobile app by storing its data online and giving you the necessary tools to access it. The framework provides everything you need right from the start.

The data of the application is split into two parts. The first part contains only the business (structured) data and the second part contains the pictures (unstructured data) used in the application.

Hosting Unstructured Data

We used Azure Blob storage for images, which provides us with a convenient way to serve them directly, thus removing the hassle of having to save images into a database.

Hosting Structured Data

On the business data side of things, we used a ASP.NET application that connects to a SQL server database. This application is deployed to Azure App Services and is accessible 24/7 from anywhere in the world. The code of the application can be found in the following GitHub repository.

The application uses Entity Framework's Code First approach to create the database. Inside the DataObjects folder, you can find the entities that the application needs. Please note that they all derive from EntityData. This will help in any data serialization/deserialization process.

As with any real-world application, the separate entities are interconnected. The customer has a collection of addresses and a collection of orders. Having such relations can result in complications when loading the required information from the database. To avoid this, we have introduced custom TableControllers that can be found inside the Controllers folder

These controllers use a custom ActionFilterAttribute to control what additional information gets loaded when you execute a query against the database.

```
public class CustomerController : TableController<Customer>

            {

                protected override void Initialize(HttpControllerContext
controllerContext)

                {

                    base.Initialize(controllerContext);
                    telerikerpContext context = new telerikerpContext();

                    DomainManager = new EntityDomainManager<Customer>
(context, Request);

                }


                // GET tables/Customer

                [ExpandProperty("Addresses")]

                public IQueryable<Customer> GetAllCustomers()

                {

                    return Query();

                }


                // GET tables/Customer/48D68C86-6EA6-4C25-AA33-223FC9A27959
                [ExpandProperty("Addresses")]

                [ExpandProperty("Orders")]

                public SingleResult<Customer> GetCustomer(string id)

                {

                    return Lookup(id);

                }

            }
```

A Deep Dive into the CustomerController

You can see that the GetAllCustomers() method has the ExpandProperty("Addresses") attribute. This will populate the addresses collection for each of the customers before returning result. The GetCustomer(string id) method will load the addresses as well as the orders of a single customer. If we didn't introduce these controllers, the Entity Framework would return empty collections for the addresses and orders and we would be forced to do additional roundtrips to the database to fetch them. Controllers allow us to do this in a single roundtrip.

To allow the Entity Framework to create, connect and modify the SQL database, we have created a custom DbContext that can be found inside the Models folder. This context holds the separate DbSets, which the application needs, and the connection string to the database. It also configures the DbModelBuilder. Based on this context and the EntityDate objects, the Entity Framework autogenerates code that is used to create the database schema. This code is then wrapped in a migration, which can be found in the Migrations folder. Executing this migration will prepare all tables in the database.

```
 public class telerikerpContext : DbContext

    {

        private const string connectionStringName = "Name=MS_
TableConnectionString";

        public telerikerpContext() : base(connectionStringName)

        {

        }

        public DbSet<Customer> Customers { get; set; }

        public DbSet<CustomerAddress> CustomerAddresses { get; set; }

        public DbSet<Product> Products { get; set; }

        public DbSet<Vendor> Vendors { get; set; }

        public DbSet<Order> Orders { get; set; }

        public DbSet<OrderDetail> OrderDetails { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)

        {

            modelBuilder.Conventions.Add(

                new AttributeToColumnAnnotationConvention<TableColumnAttribu
te, string>(

                    "ServiceTableColumn", (property, attributes) =>
attributes.Single().ColumnType.ToString()));

        }

    }
```

The final step is to populate the database with actual data. This is achieved by a custom DbMigrationsConfiguration, which allows us to override the Seed() method and control what data gets populated into the separate database tables.

```
internal sealed class Configuration : DbMigrationsConfiguration<telerikerpSe
rvice.Models.telerikerpContext>

{

    public Configuration()

    {

        AutomaticMigrationsEnabled = false;

        SetSqlGenerator("System.Data.SqlClient",
new EntityTableSqlGenerator());

        ContextKey = "telerikerpService.Models.telerikerpContext";

    }

    protected override void Seed(telerikerpService.Models.telerikerpContext
context)

    {

        //  This method will be called after migrating to the latest version.

        if (!context.Customers.Any())

        {

            context.Customers.AddOrUpdate(SeedData.Customers);
            context.CustomerAddresses.AddOrUpdate(SeedData.CustomerAddresses);

        }

        if (!context.Products.Any())

        {

            context.Products.AddOrUpdate(SeedData.Products);

        }

        if (!context.Vendors.Any())

        {

            context.Vendors.AddOrUpdate(SeedData.Vendors);

        }

        if (!context.Orders.Any())

        {
            context.Orders.AddOrUpdate(SeedData.Orders);
            context.OrderDetails.AddOrUpdate(SeedData.OrderDetails);
       }
    }

}
```

Note that the actual initial data is stored in a static class, which can be found in the App_Start folder.

Now that we covered the backend, let's continue with the application structure. We will go over the UI and the MVVM framework of choice.

Telerik ERP: MVVM Framework and UIs

Telerik ERP is built using the Xamarin.Forms cross-platform mobile framework and the MVVM (Model-View-ViewModel) pattern. While Xamarin.Forms itself offers some features to get you started with MVVM, there are other libraries that will help you structure and optimize the code in an even better way. One such framework is MvvmCross. It is specifically developed for Xamarin and the mobile ecosystem.

In addition to all the features provided by Xamarin.Forms, MvvmCross provides functionalities like:

- ViewModel to ViewModel navigation—a support for passing and returning objects
- Built-in dependency injection, bindings and inversion of control
- Extensive messaging
- Many classes that help building the application
- Ability to change the native platform navigation

When building our ERP sample application, we made sure to stick to the best programming practices by the worldwide Xamarin community. Most of you will relate to building applications with MvvmCross.

On top of the code implementation sits the beautiful and highly-performant UI of the application, built with Telerik UI for Xamarin controls.

ERP MvvmCross Implementation

The starting point of the ERP application is the ErpApplication class. This class inherits from the MvxApplication—a class provided by the MvvmCross framework. The main idea behind the custom application is to add additional configuration before it starts. For the ERP application, all services get registered:

```
public class ErpApplication : MvxApplication

            {

                public override void Initialize()

                {

                    CreatableTypes()

                        .InNamespace(typeof(Services.IErpService).Namespace)

                        .EndingWith("Service")

                        .AsInterfaces()

                        .RegisterAsLazySingleton();

                    RegisterCustomAppStart<AppStart>();

                }

            }
```

Services

The AuthenticationService is used by the LoginPageViewMode. It takes the provided username and password and validates them. If the authentication is successful, the main page is visualized. Currently, the service does not have a specific implementation while login is executed. It is up to you to implement the desired functionality here:

```
public class AuthenticationService : IAuthenticationService

    {

        public string UserName { get; private set; }

        public Task<bool> IsAuthenticated()

        {

            // TODO: Cache authentication

            return Task.FromResult(false);

        }
        public Task<bool> Login(string username, string password)

        {

            this.UserName = username;

            // allows login with random username and password combination

            return Task.FromResult(true);

        }

    }
```

The ErpService then establishes the connection with the database. Its main purpose is to execute the CRUD operations with the data provided by the database. Through its constructor, an IMvxMessenger is injected.

```
public ErpService(IMvxMessenger messenger)

                {

                    this.messenger = messenger;
```

This is a specific service provided by MvvmCross and its main purpose is to send messages to the ViewModels. In order to receive the messages, the ViewModel should inject the service as well, which enables you to subscribe to messages sent by the ErpService. Each ViewModel must call one of the subscribe methods on the IMvxMessenger and must store the returned token in order to be able to successfully receive the messages.

ViewModels

A constructor injection is used for each ViewModel. This injection is used internally within MvvmCross when the ViewModels are created. The base class of each ViewModel is the MvcViewModel class. By inheriting from it, the services could be provided to each ViewModel through injection.

ViewModels are essential to the navigation in Xamarin.Forms because ViewModel-First is the default navigation pattern in MvvmCross. This means that we navigate from ViewModel to ViewModel and not from View to View. It ships with its own navigation system called IMvxNavigationService to navigate between these ViewModels. Almost every ViewModel has an injected IMvxNavigationService service through its constructor.

Let's look at one of the ViewModels that is part of the ERP application—the DashboardPageViewModel.

```
public class DashboardPageViewModel : MvxViewModel

           {

               public DashboardPageViewModel(Services.IErpService service,
IMvxNavigationService navigationService)

               {

                   this.service = service;

                   this.navigationService = navigationService;

                   this.AboutCommand = new MvxCommand(ShowAboutPage);

               }
```

As explained above, the ViewModel inherits from the MvxViewModel and, through its constructor, an ErpService and IMvxNavigationService are injected. Inside the constructor, only some fields and properties are assigned. The data visualized in the View part is fetched using a specific method called Prepare. The method is part of the base MvxViewModel. This is the initial point of the ViewModel. You can use this method to receive and store parameters. It is called every time a navigation is executed to the ViewModel. Here, the data is fetched:

```
public async override void Prepare()

               {

                   base.Prepare();

                   IsBusy = true;

                   await this.FetchData();

                   if (await this.service.SyncIfNeededAsync())

                   {

                       await FetchData();

                   }

                   IsBusy = false;

               }
```

From this ViewModel, navigation is only executed to the AboutPage. For this purpose, the injected navigation service is used, and navigation happens from one ViewModel to another:

```
private void ShowAboutPage()
                {
                    this.navigationService.Navigate<AboutPageViewModel>();
                }
```

Views

MvvmCross uses ViewModel-first navigation, meaning it navigates from ViewModel to ViewModel and not from View to View. This is achieved by following a naming convention.

For the ERP application, however, we used another feature provided by MvvmCross—the generic implementation of the MvxContextPage where you can specify the exact ViewModel of the page.

Each View should inherit from the MvxContextPage instead of the Xamarin.Forms content page. Here is how the DashboardPage inherits from MvxContentPage:

```
public partial class DashboardPage :
MvxContentPage<ViewModels.DashboardPageViewModel>,
IMvxOverridePresentationAttribute
```

Here, the ViewModel of the page is passed using the generic version of the MvxContentPage. There are some built-in attributes through which you can define how a View will be displayed. Some of the existing attributes are used to tell the page that it is a Modal or is inside a NavigationView.

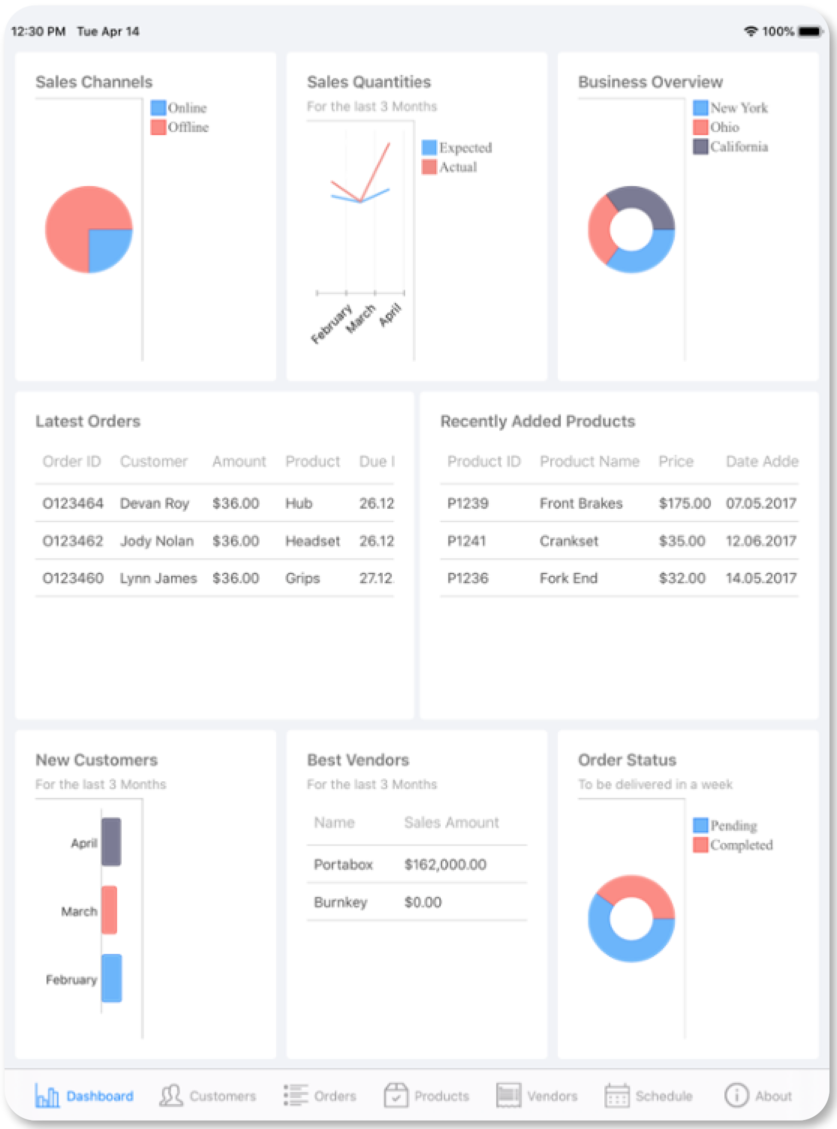
The Telerik ERP application has a unique look for desktop, tablet and phone that is achieved using custom presentation attributes. By implementing the IMvxOverridePresentationAttribute, you can set a specific custom attribute for the page. For the DashboardPage, for example, if the application is visualized on phone, the page is wrapped inside a navigation page. When displayed on desktop or tablet, it isn't:

```
public MvxBasePresentationAttribute PresentationAttribute(MvxViewModel
Request request)

                {

                    if (Device.Idiom == TargetIdiom.Phone)

                    {

                        return new MvxTabbedPagePresentationAttribute
(TabbedPosition.Tab) { WrapInNavigationPage = true };

                    }

                    else

                    {

                        return new MvxTabbedPagePresentationAttribute
(TabbedPosition.Tab) { WrapInNavigationPage = false };

                    }

                }
```

In addition to the different wrapping, the page itself has a different look for desktop and Phone:

```
<mcf:MvxContentPage.Content>

                <OnIdiom x:TypeArguments="ContentView">

                    <OnIdiom.Phone>

                        <cards:PhoneView/>

                    </OnIdiom.Phone>

                    <OnIdiom.Tablet>

                        <cards:TabletView />

                    </OnIdiom.Tablet>

                    <OnIdiom.Desktop>

                        <cards:TabletView />

                    </OnIdiom.Desktop>

                </OnIdiom>

</mcf:MvxContentPage.Content>
```

Here is how the page is visualized on both iPad and iPhone:



Latest Orders

Order ID	Customer	Amount	Product	Due I
0123464	Devan Roy	$36.00	Hub	26.12
0123462	Jody Nolan	$36.00	Headset	26.12
0123460	Lynn James	$36.00	Grips	27.12

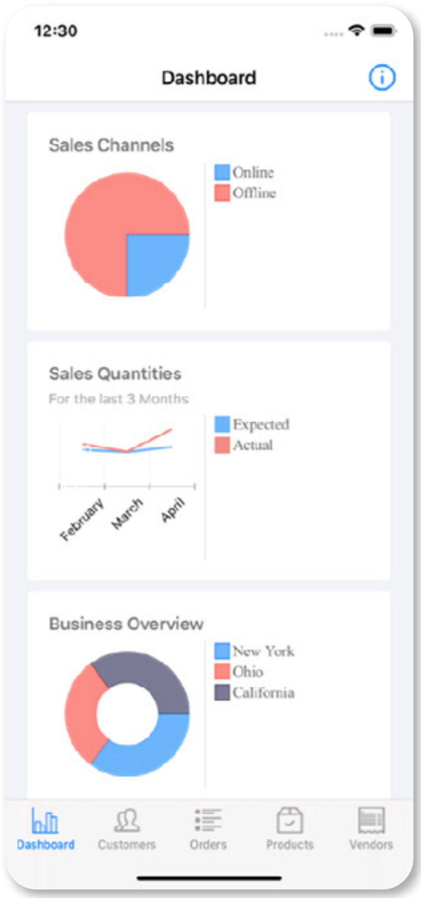
Recently Added Products

Product ID	Product Name	Price	Date Adde
P1239	Front Brakes	$175.00	07.05.2017
P1241	Crankset	$35.00	12.06.2017
P1236	Fork End	$32.00	14.05.2017

Best Vendors

For the last 3 Months

Name	Sales Amount
Portabox	$162,000.00
Burnkey	$0.00



Views UI

The main heroes of the DashboardPage are two Telerik UI for Xamarin controls—the Chart and DataGrid.

Xamarin Charts

Telerik UI for Xamarin provides many different types of charts and series that are extremely flexible. Some of them are presented and visualized in the DashboardPage.

With the Chart control, you can show trends and data using line charts, area charts and spline charts, or visualize data comparisons using bar charts. You can even depict proportions with pie charts. The PieChart is the main hero in the DashboardPage.

The Sales Channels Data is presented using the PieChart. The data is provided to the PieChart using the PieSeries. The series only need to have a provided ItemsSource in order to be able to visualize the data. The exact data comes from the DashboardPageViewModel, which fetches its data when the Prepare method is executed:

```
<telerikChart:RadPieChart x:Name="pieChart" Grid.Row="1" Grid.Column="0">

            <telerikChart:RadPieChart.ChartBehaviors>

                <telerikChart:ChartSelectionBehavior
SeriesSelectionMode="None" DataPointSelectionMode="None"/>

            </telerikChart:RadPieChart.ChartBehaviors>

            <telerikChart:RadPieChart.Series>

                <telerikChart:PieSeries ItemsSource="{Binding SalesChannels}"
ShowLabels="false" ValueBinding="Value" LegendTitleBinding="Name"
LabelBinding="Name"/>

            </telerikChart:RadPieChart.Series>

        </telerikChart:RadPieChart>
```

The legend displayed on the right side of the Xamarin PieChart is a separate control. The control makes it easy for you to provide a description regarding the series that are visualized within the chart. The ChartProvider should only be set to the chart control whose series will be described in the legend and the legend will visualize the items out of the box. Using the LegendTitleBinding of the series, you can specify the item that will be used as a title for the legend.

```
<telerikChart:RadLegend Grid.Row="1" Grid.Column="1" HeightRequest="200"

                            Style="{StaticResource LegendStyle}"

                            LegendProvider="{x:Reference Name=pieChart}"/>
```

The Sales Quantities data is presented using another type of Xamarin Chart—the CartesianChart. Here, we have two LineSeries that visualize the expected and actual sales data. The visualization of several types of series is supported out-of-the-box by the Chart control. For the CartesianChart, we have specified both the horizontal and vertical axes as well. The CartesianChart plots data points in a coordinate system defined by its two axes. Each axis provides information about the values of the data points using ticks and text labels. A legend is displayed on the right side of the chart.

```
<telerikChart:RadCartesianChart x:Name="chart" Grid.Row="2" Grid.Column="0">

            <telerikChart:RadCartesianChart.ChartBehaviors>

                <telerikChart:ChartSelectionBehavior
SeriesSelectionMode="None" DataPointSelectionMode="None"/>

            </telerikChart:RadCartesianChart.ChartBehaviors>

            <telerikChart:RadCartesianChart.Series>

                <telerikChart:LineSeries ItemsSource="{Binding
ExpectedSalesQuantitues}" ShowLabels="false" DisplayName="Expected">

                    <telerikChart:LineSeries.ValueBinding>

                        <telerikChart:PropertyNameDataPointBinding
PropertyName="Value"/>

                    </telerikChart:LineSeries.ValueBinding>

                    <telerikChart:LineSeries.CategoryBinding>

                        <telerikChart:PropertyNameDataPointBinding
PropertyName="Name" />

                    </telerikChart:LineSeries.CategoryBinding>

                </telerikChart:LineSeries>

                <telerikChart:LineSeries ItemsSource="{Binding
ActualSalesQuantitues}" ShowLabels="false" DisplayName="Actual">

                    <telerikChart:LineSeries.ValueBinding>

                        <telerikChart:PropertyNameDataPointBinding
PropertyName="Value"/>

                    </telerikChart:LineSeries.ValueBinding>

                    <telerikChart:LineSeries.CategoryBinding>

                        <telerikChart:PropertyNameDataPointBinding
PropertyName="Name" />

                    </telerikChart:LineSeries.CategoryBinding>

                </telerikChart:LineSeries>

            </telerikChart:RadCartesianChart.Series>
            <telerikChart:RadCartesianChart.HorizontalAxis>

                <telerikChart:CategoricalAxis LabelTextColor="Black"
LabelFitMode="Rotate" />

            </telerikChart:RadCartesianChart.HorizontalAxis>

            <telerikChart:RadCartesianChart.VerticalAxis>

                <telerikChart:NumericalAxis LabelTextColor="White"
Minimum="0" />

            </telerikChart:RadCartesianChart.VerticalAxis>

            <telerikChart:RadCartesianChart.Grid>

                <telerikChart:CartesianChartGrid MajorLinesVisibility="X" />

            </telerikChart:RadCartesianChart.Grid>

        </telerikChart:RadCartesianChart>
```

The Business Overview and Order Status data is visualized with RadPieChart. This time we used DonutSeries instead of PieSeries. DonutSeries visualizes the chart in the shape of a donut. The inner empty space is set according to the InnerRadiusFactor property. Each data item is visually represented by a donut slice. There are no other differences compared to the PieChart visualized in the Sales Channels.

Another series, the BarSeries, visualizes the New Customers data.

You can find detailed information about the Chart control and all the series and axes it provides in the official documentation of the Telerik UI for Xamarin controls.

Xamarin DataGrid

The tables that are visualized on the page are implemented using yet another control from the Telerik UI for Xamarin suite—the Xamarin DataGrid control.

Most of the data on the Internet is stored in tables within a database. RadDataGrid for Xamarin provides the same abstraction over the data. It has columns and rows and the intersection of a row and a column—a cell.. The control supports different types of columns, selection modes, loading data on demand, a rich customization API and many more features.

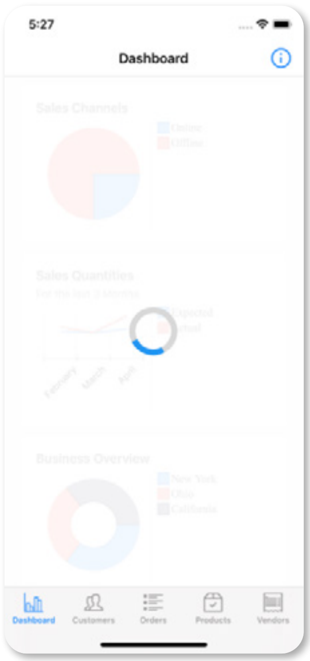
In order to visualize data using the DataGrid control, you have to set its ItemsSource property. By default, the control will generate out-of-the-box columns using the provided data type. If you want to manually declare the desired columns, set the AutoGenerateColumns property to false.

In the DashboardPage, the DataGrid control is used to visualize data for the latest orders, recently added products and best vendors.

Here is how the DataGrid is declared for the best vendors part:

```
<telerikDataGrid:RadDataGrid Grid.Row="2" Grid.ColumnSpan="2" 
ItemsSource="{Binding BestVendors}"

                                     AutoGenerateColumns="false" 
Style="{StaticResource DataGridStyle}">

            <telerikDataGrid:RadDataGrid.Columns>

                <telerikDataGrid:DataGridTextColumn HeaderText="Name" 
PropertyName="Name" HeaderStyle="{StaticResource cellHeaderStyle}" />

                <telerikDataGrid:DataGridTextColumn HeaderText="Sales 
Amount" PropertyName="SalesAmount" CellContentFormat="{}{0:C}" 
HeaderStyle="{StaticResource cellHeaderStyle}" />

            </telerikDataGrid:RadDataGrid.Columns>

        </telerikDataGrid:RadDataGrid>
```

You can find more information about the DataGrid control and its features in our official documentation.

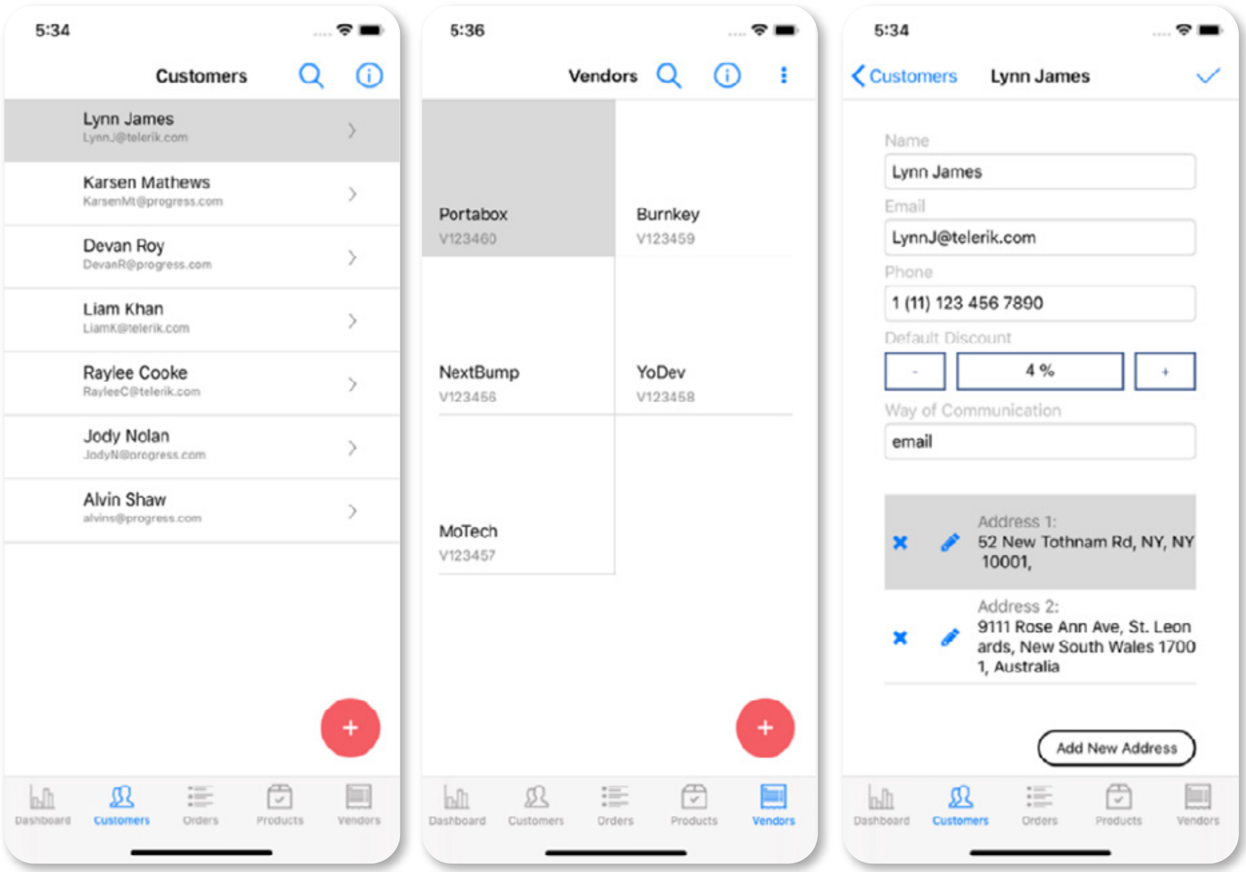


Xamarin BusyIndicator

Telerik ERP uses the beautiful Telerik UI for Xamarin BusyIndicator control to illustrate that a long-running process is executing in the background. The BusyIndicator control supports ten different animations (by the time of writing) right out of the box and can be customized with visuals of your choice. Here is what we selected for the ERP application:

Advanced Xamarin ListView

RadListView is one of the most popular Telerik UI For Xamarin controls that is used by the ERP application to visualize data. You can bind it to a data source and display templated collections of items in a nicely formatted list. It supports different types of layouts, e.g. you can show the data as a vertical list or visualize the items in rows and columns using the ListViewGridLayout mode. This control offers a great variety of built-in features like grouping, sorting, filtering, reordering, item swipe, load on demand functionality (it speeds the initial loading of the page by rendering items that are visible before fetching the remaining items from the data source), great customization APIs and many more.



Lynn James
LynnJ@telerik.com
Karsen Mathews
KarsenMt@progress.com
Devan Roy
DevanR@progress.com
Liam Khan
LiamK@telerik.com
Raylee Cooke
RayleeC@telerik.com
Jody Nolan
JodyN@progress.com
Alvin Shaw
alvins@progress.com

Portabox V123460	Burnkey V123459
NextBump V123456	YoDev V123458
MoTech V123457	

Name: Lynn James
Email: LynnJ@telerik.com
Phone: 1 (11) 123 456 7890
Default Discount: 4 %
Way of Communication: email

Address 1:
52 New Tothnam Rd, NY, NY 10001,

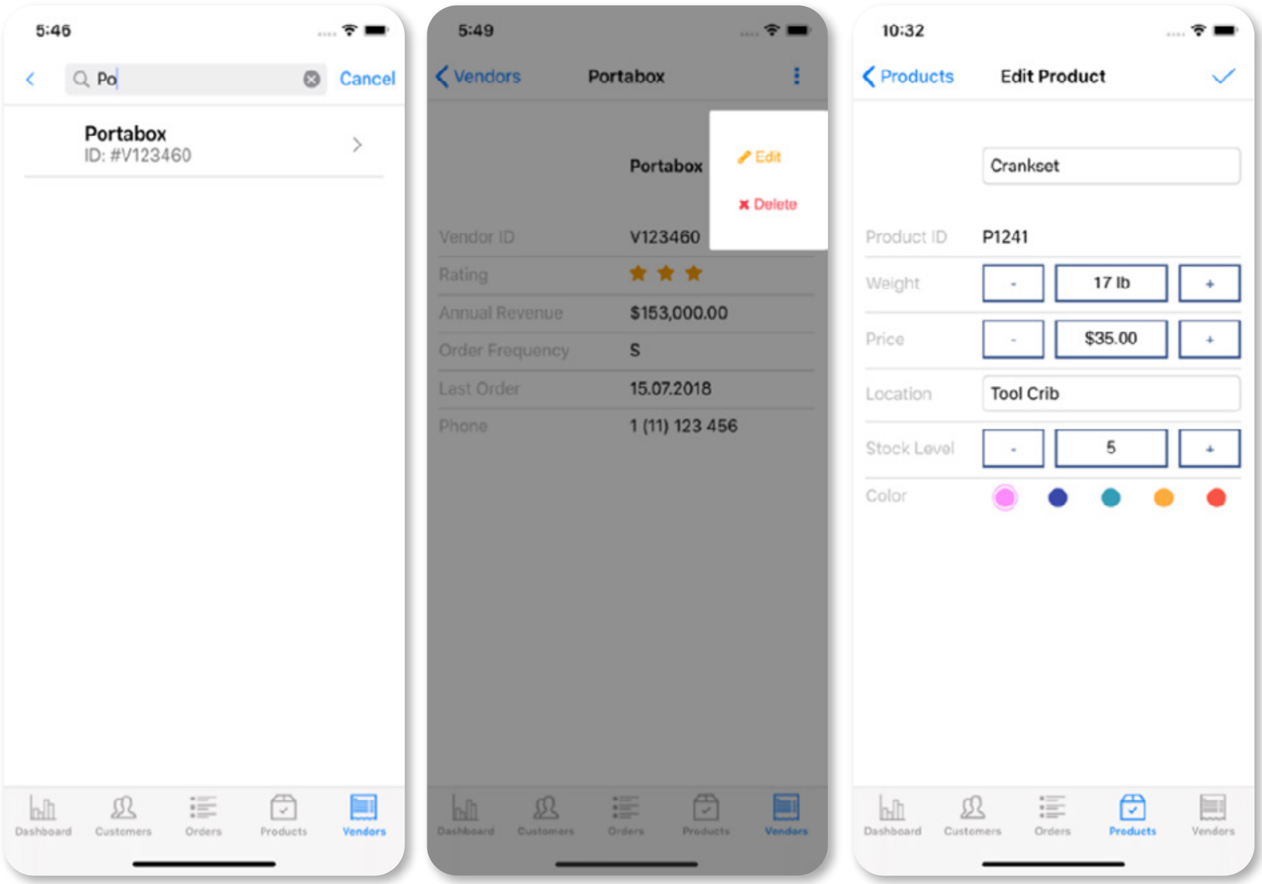
Address 2:
9111 Rose Ann Ave, St. Leonards, New South Wales 1700 1, Australia

The filtering functionality of the RadListView is demonstrated by the ERP application in combination with another Telerik UI for Xamarin control called RadEntry.

The Telerik Entry control provides the core functionality expected by any entry in addition to many more, including the option to customize the border around the input using the BorderStyle property.

The Xamarin Popup control is used to visualize dialogs with additional content on top of other controls. The component provides a flexible API to easily control its behavior and makes possible the implementation of complex logic for a wide range of scenarios. You can make it modal (that makes the UI behind it inactive) if you need to or change the animation that is used when the control is visualized and gets closed.

Many other Telerik UI for Xamarin controls like RadNumericInput, RadBorder, RadButton and RadPath are used by the pages.



Portabox
ID: #V123460

Portabox

Vendor ID V123460
Rating ★ ★ ★
Annual Revenue $153,000.00
Order Frequency S
Last Order 15.07.2018
Phone 1 (11) 123 456

Crankset
Product ID P1241
Weight - 17 lb +
Price - $35.00 +
Location Tool Crib
Stock Level - 5 +
Color ● ● ● ● ●

The Telerik UI for Xamarin.Forms NumericInput control enables you to set or edit a specific double number by using either the input field or the increase and decrease buttons. The buttons used by the NumericInput are RadButtons. RadButton provides all the core features expected by a button control, plus additional features like content alignment, background image support, changing the button shape and more. You can also set the appearance of a button's border with the RadBorder control. With it, you can style the border thickness and change the corner radius of each Xamarin.Forms control.

Using Xamarin.Forms in combination with MvvmCross and the Telerik UI for Xamarin controls, we were able to build a beautiful, fast and feature-rich application that enables you to manage and grow your business with ease from every corner of the planet.

Install Telerik UI for Xamarin Controls

Get Application Source Code

Download Telerik ERP demo application for iOS, Android and Windows devices.

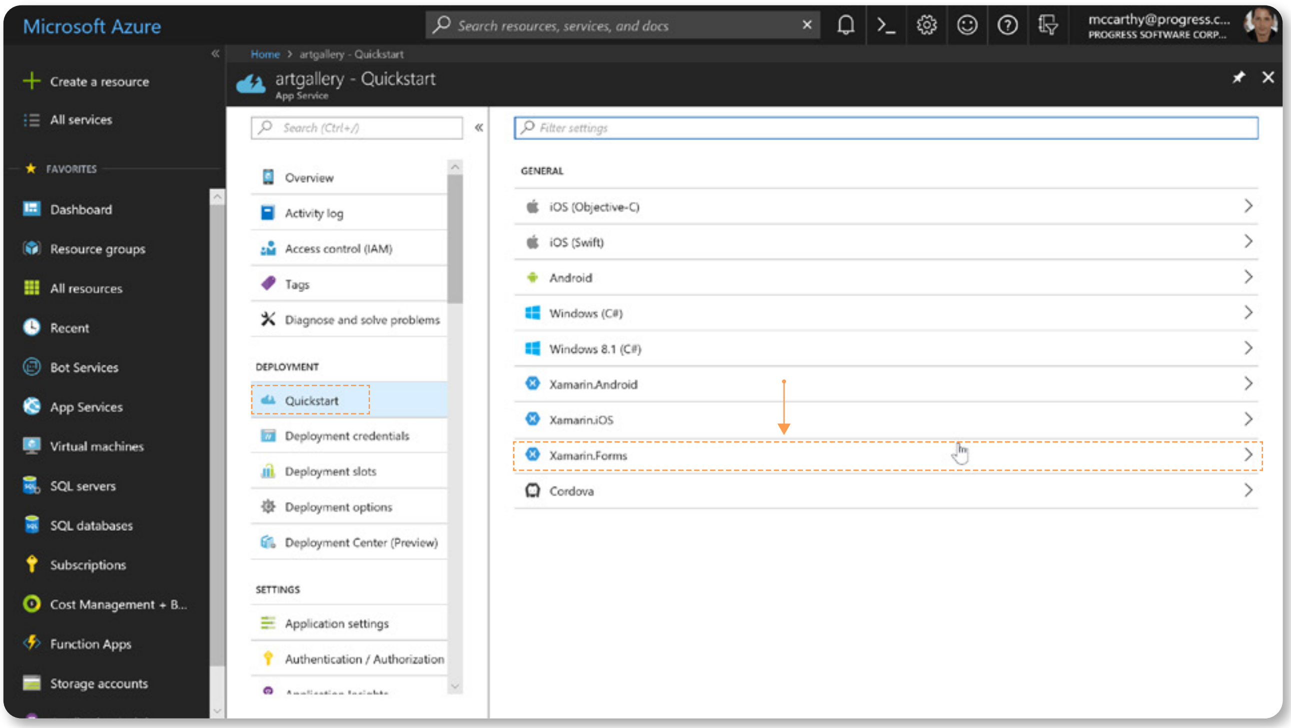
Telerik CRM: How to Build a Mobile Customer Relationship Management Application with Xamarin.Forms

Azure App Service Walkthrough

Microsoft recently updated the [Azure App Service documentation with a Xamarin.Forms tutorial.]() It walks you through initializing the App Service and creating the database the service will use. Let's go through the process together.

Configuration and Starter Projects

After creating the service, go to the Quickstart tab and you will be presented with a list of client application types. Select the Xamarin.Forms item.



A new Azure blade will appear. Follow these steps:

Step 1: Create the database

If you already have a SQL Server and/or SQL Database, select this option. Alternatively, you can always choose to create a new server and database.

Note: When creating the SQL Server, you will need to create admin credentials. You don't have to remember them for the App Service since they are stored in an environment variable, but it might be a good idea to store them in your preferred password manager or another secure location.

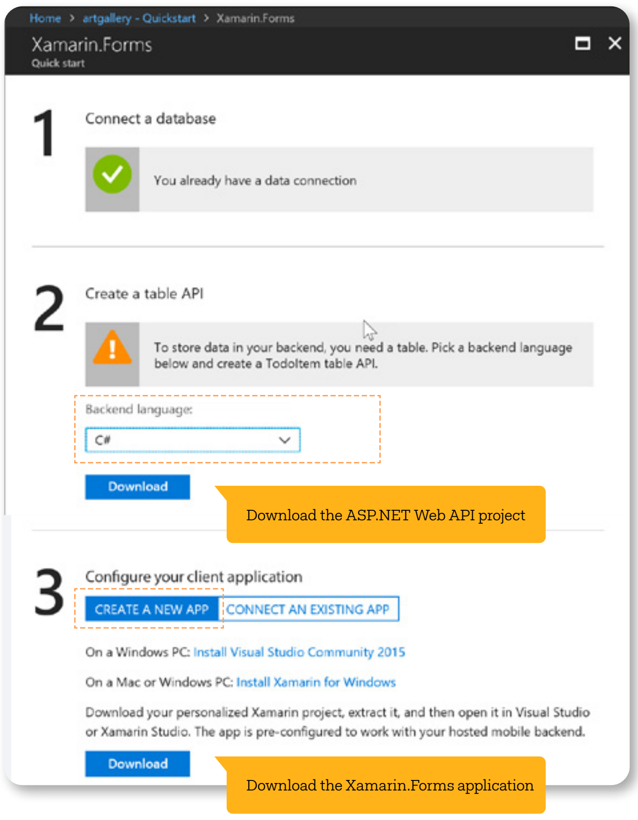
After you make your selection for a C# backend, click the Download button. Azure will create an ASP.NET project for you, already set up with the right connection string to the database you've selected at Step 1.

Step 2: Select Xamarin.Forms project

This step gives you two options: "create a new app" or "connect an existing app". If you choose to connect an existing app, you'll be given a few code snippets and instructions.

For Telerik CRM, we chose to create a new app. As with the backend, when you click the Download button, you'll get a new Xamarin.Forms application that already has the required code to connect to the App Service.

Here's a screenshot of the "Xamarin.Forms Quick start" blade:


Home > artgallery - Quickstart > Xamarin.Forms

Xamarin.Forms
Quick start

1 Connect a database

You already have a data connection

2 Create a table API

To store data in your backend, you need a table. Pick a backend language below and create a TodoItem table API.

Backend language: C#

Download

Download the ASP.NET Web API project

3 Configure your client application

CREATE A NEW APP | CONNECT AN EXISTING APP

On a Windows PC: Install Visual Studio Community 2015

On a Mac or Windows PC: Install Xamarin for Windows

Download your personalized Xamarin project, extract it, and then open it in Visual Studio or Xamarin Studio. The app is pre-configured to work with your hosted mobile backend.

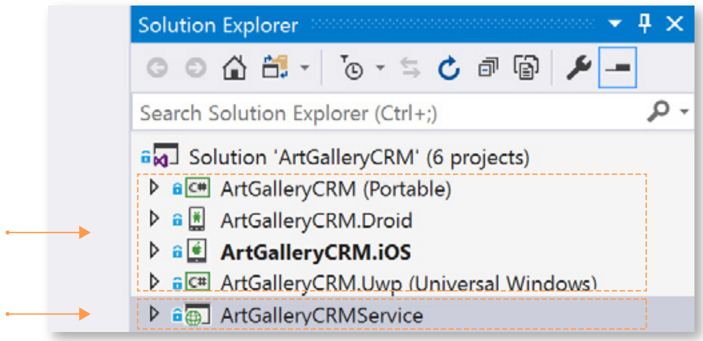
Download

Download the Xamarin.Forms application

When done, you will have downloaded two ZIP files, containing two solutions—a Xamarin. Forms solution and an ASP.NET MVC solution. These are your starter projects.

If you have already checked the source code of the Telerik CRM app, you will notice that we combined all the projects into a single folder and created a combined solution. This is not necessary. We did it to keep all the projects for this tutorial in one place for simpler source control and to make the tutorial easier to follow.

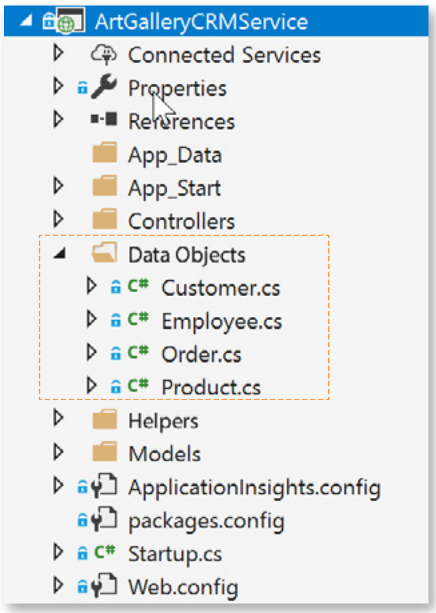
Here is the result: The arrows point to the two different downloads combined in a single solution.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ArtGalleryCRM' (6 projects)
ArtGalleryCRM (Portable)
ArtGalleryCRM.Droid
ArtGalleryCRM.iOS
ArtGalleryCRM.Uwp (Universal Windows)
ArtGalleryCRMService

Moving on to the code, let's start with the ASP.NET service project. We want to complete the backend before we add the UI.

Entities

In the project, you will see a DataObjects folder, containing a TodoItem.cs class. We need to replace it with the data models necessary for the Art Gallery CRM app.



These are the entities we need:

- Employee
- Customer
- Product
- Order

We created four classes to define the entity model and its properties.

Note: If you are not familiar with entity frameworks and code-first approach, you can use the existing TodoItem.cs class to help guide you before deleting it.

Each of the classes have properties related to the entity. These properties will be used to create the table fields. For example, an Employee would have OfficeLocation, while an Order may need properties like ProductId and Quantity.

```
public class Employee : EntityData

    {
        public string Name { get; set; }

        public string PhotoUri { get; set; }

        public string OfficeLocation { get; set; }

        public DateTime HireDate { get; set; }

        public double Salary { get; set; }

        public int VacationBalance { get; set; }

        public int VacationUsed { get; set; }

        public string Notes { get; set; }
    }
```

It is important that you finalize the properties at this stage because, once you initialize the database, Entity Framework will create the database fields using those properties. Any changes to the model will require a special operation called Entity Framework Code First Migration, which can quickly get complicated.

DbContext

We now need to update the database context class used by Entity Framework to interact with the database. Go ahead and replace the TodoItem DbSet with the four new ones:

```
public class ArtGalleryCRMContext : DbContext

    {
        private const string connectionStringName = “Name=MS_
TableConnectionString”;

        public ArtGalleryCRMContext() : base(connectionStringName)

        {
        }

        public DbSet<Employee> Employees { get; set; }

        public DbSet<Customer> Customers { get; set; }

        public DbSet<Product> Products { get; set; }

        public DbSet<Order> Orders { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)

        {

            modelBuilder.Conventions.Add(new AttributeToColumnAnnotationConve
ntion<TableColumnAttribute, string>(

                "ServiceTableColumn",

                (property, attributes) => attributes.Single().ColumnType.
ToString()));

        }

    }
```

Database Initializer

In the AppStart folder, you will find that the autogenerated project nested a class that seeds the database with the tables and initial records. We moved that class into its own class and replaced the TodoItem seed data with records for each of the entities:

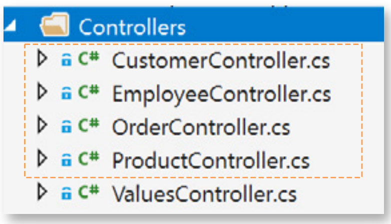
```
// Seeds the database with tables for the related entities.

    public class ArtGalleryCRMInitializer : CreateDatabaseIfNotExists
<ArtGalleryCRMContext>

    {

        protected override void Seed(ArtGalleryCRMContext context)

        {

            // Generate Employees Customers and Products data

            var employees = SampleDataHelpers.GenerateEmployees();

            var customers = SampleDataHelpers.GenerateCustomers();

            var products = SampleDataHelpers.GenerateProducts();

            var orders = SampleDataHelpers.GenerateOrders(employees,
customers, products);
```

```
            // Seed the Database tables

            foreach (var employee in employees) context.Set<Employee>().
Add(employee);

            foreach (var customer in customers) context.Set<Customer>().
Add(customer);

            foreach (var product in products) context.Set<Product>().
Add(product);

            foreach (var order in orders) context.Set<Order>().Add(order);

            base.Seed(context);

        }

    }
```

Table Controllers

Finally, you want a controller for each of the entities to interact with the database. This is the endpoint that the client app will use to interact with the databases and each implements the Azure Mobile Server TableController<T> interface.

We created a controller for each of the entities:


Controllers
CustomerController.cs
EmployeeController.cs
OrderController.cs
ProductController.cs
ValuesController.cs

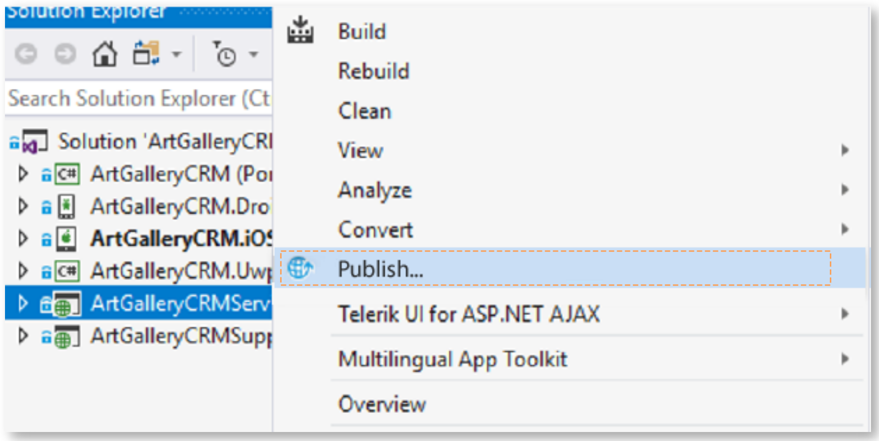
The content of each Web API controller follows the CRUD pattern you would expect. Here is a screenshot of the EmployeeController's methods (all the controllers reuse this pattern).

```
public class EmployeeController : TableController<Employee>

    {

        protected override void Initialize(HttpControllerContext
controllerContext)

        {
            base.Initialize(controllerContext);
            var context = new ArtGalleryCRMContext();
            DomainManager = new EntityDomainManager<Employee>(context,
Request);

        }
        // GET tables/Employee
        public IQueryable<Employee> GetAllEmployees()

        {

            return Query();

        }

        // GET tables/Employee/48D68C86-6EA6-4C25-AA33-223FC9A27959
        public SingleResult<Employee> GetEmployee(string id)

        {

            return Lookup(id);

        }

        // PATCH tables/Employee/48D68C86-6EA6-4C25-AA33-223FC9A27959
       public Task<Employee> PatchEmployee(string id, Delta<Employee> patch)

        {
            // Remove this before deploying to your Azure account.
               throw new HttpException(500, “Demo - Read-only mode”);

            // Uncomment before deploying to your Azure account.

            // return UpdateAsync(id, patch);
        }

        // POST tables/Employee

        public async Task<IHttpActionResult> PostEmployee(Employee employee)

        {

            // Remove this before deploying to your Azure account.
               throw new HttpException(500, “Demo - Read-only mode”);

            // Uncomment before deploying to your Azure account.

            // Employee current = await InsertAsync(employee);

            // return CreatedAtRoute(“Tables”, new { id = current.Id },
current);

        }

        // DELETE tables/Employee/48D68C86-6EA6-4C25-AA33-223FC9A27959
        public Task DeleteEmployee(string id)

        {
            throw new HttpException(500, “Demo - Read-only mode”);
            // Uncomment before deploying to your Azure account.
            // return DeleteAsync(id);
       }
    }
```

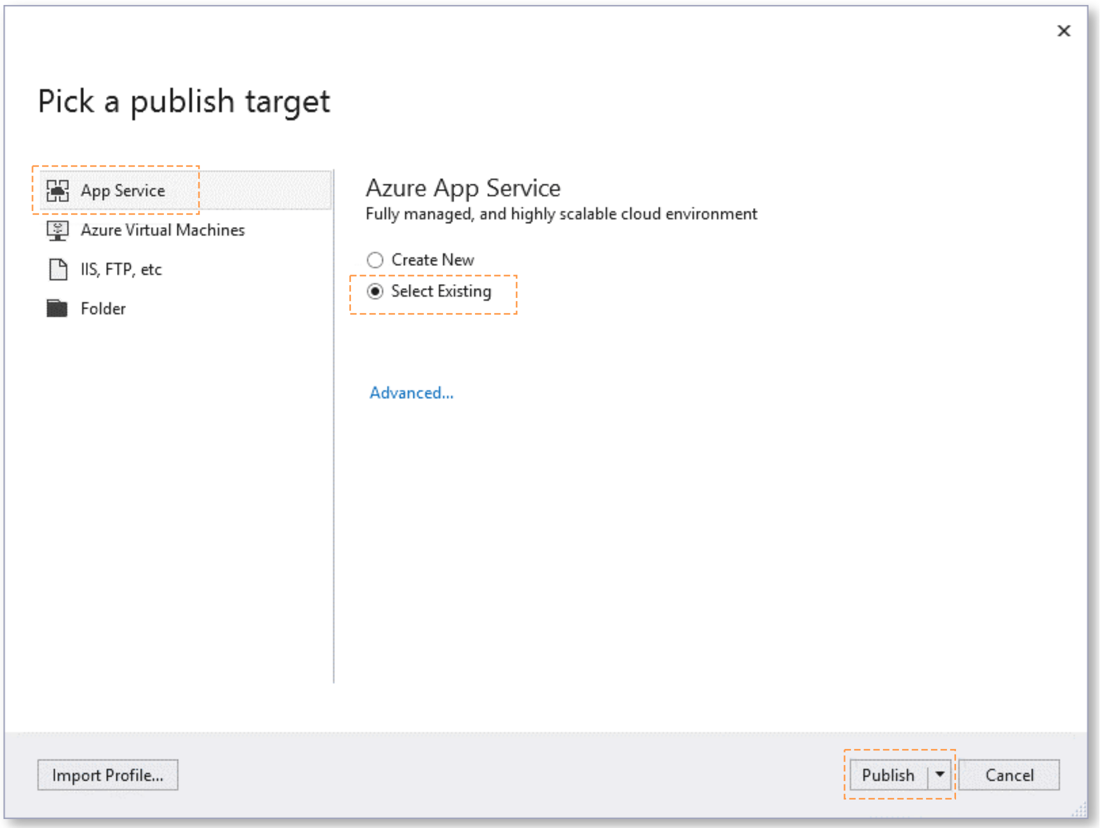
Note: If you are not familiar with ASP.NET or Web API, you can use the ToDo controller for guidance before deleting it.

Publish to Azure

Visual Studio has great built-in Azure tools. If you right-click on the ASP.NET project, you will see a “Publish” option in the context menu:



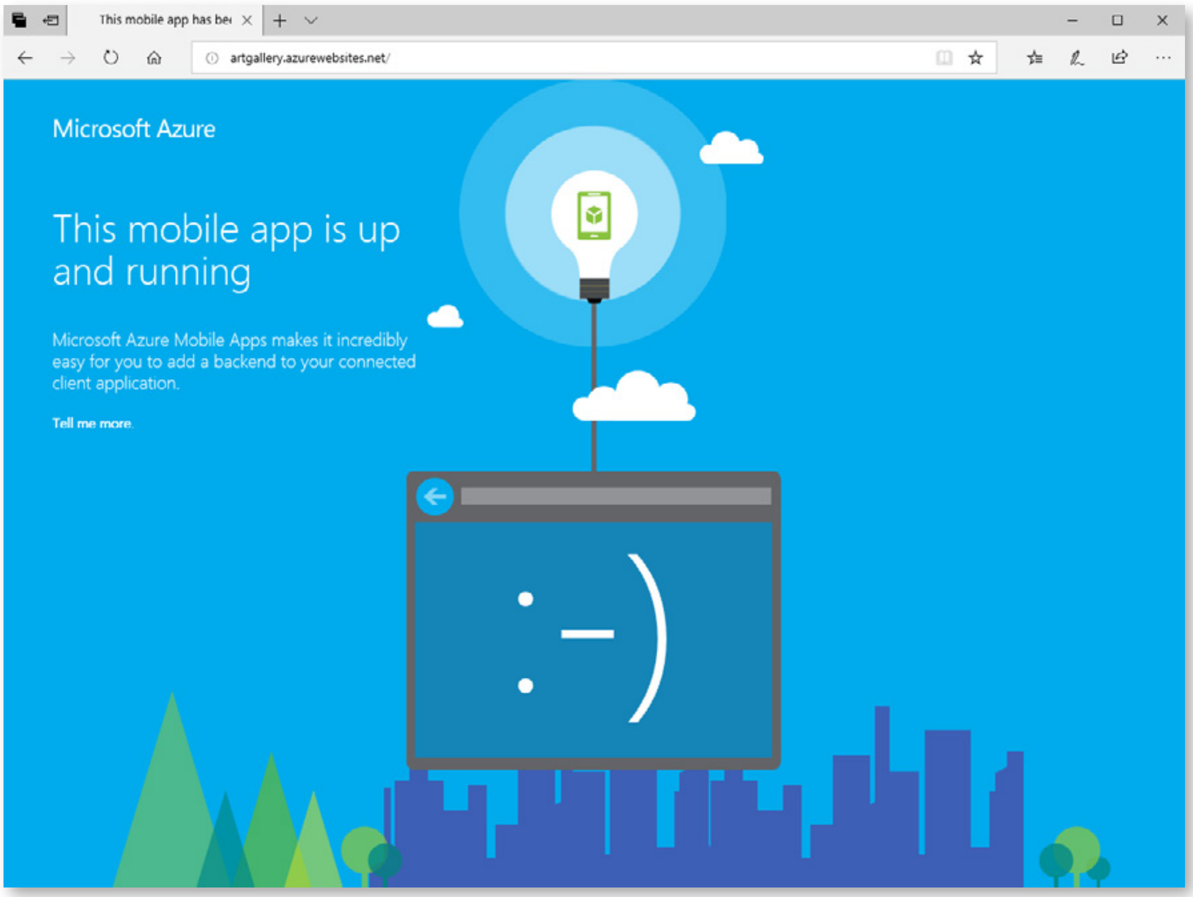
Selecting it will provide you with a list of options to choose from. Select “Azure App Service” and then “Select Existing” since we already set up the App Service.



In the next few steps, you'll be able to choose the Resource and App Service from the Azure Subscription.

Note: You will need to be signed in the same account that the Azure Subscription uses. If you have multiple accounts, there is a dropdown at the top right to change accounts.

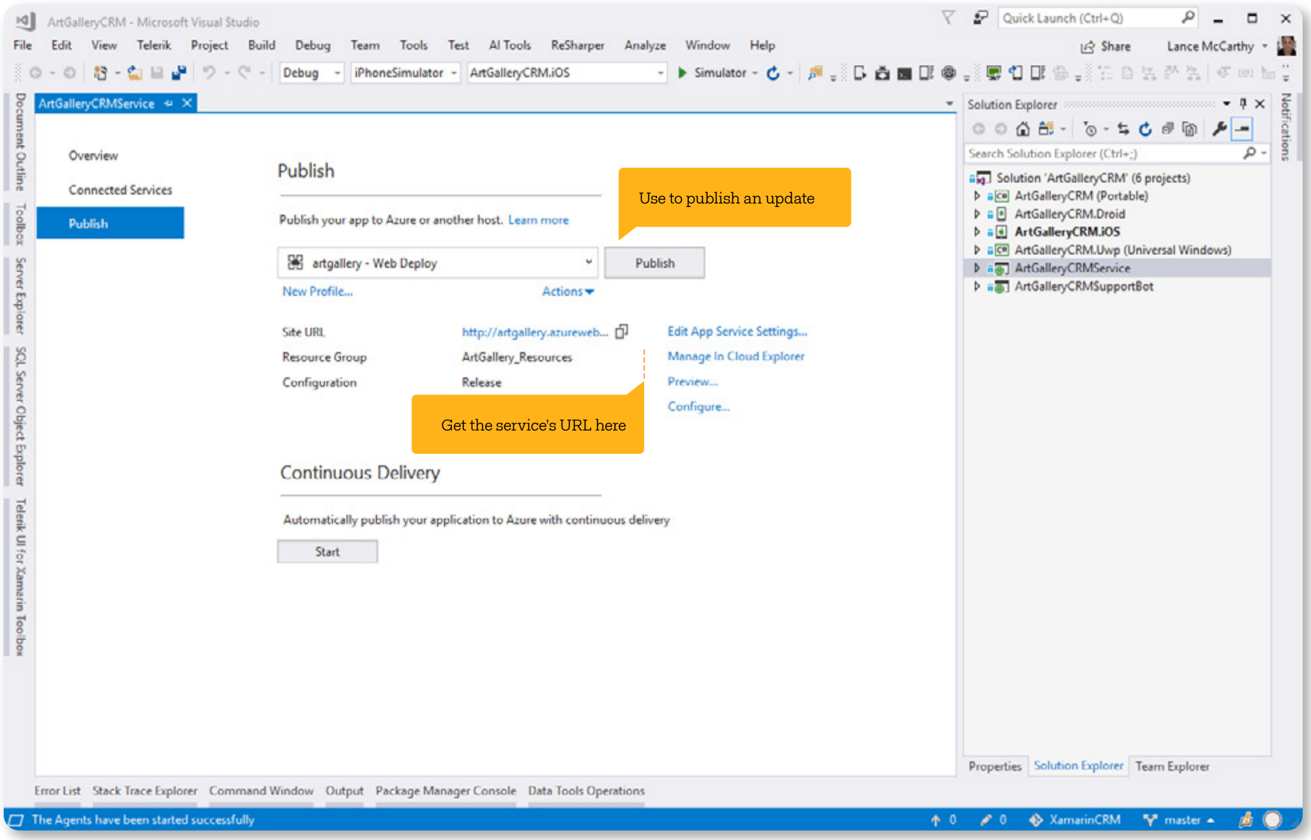
Once you've gone through the publishing steps, Visual Studio will start the publishing process and upload the application. When done, a browser tab will open and present the landing page of the running app.



This mobile app is up and running

Microsoft Azure Mobile Apps makes it incredibly easy for you to add a backend to your connected client application.

The next time you use the Publish menu option, you'll see the existing publish profile details along with several other tools:



Publish

Publish your app to Azure or another host. Learn more

artgallery - Web Deploy

Site URL http://artgallery.azureweb...
Resource Group ArtGallery_Resources
Configuration Release

Continuous Delivery

Automatically publish your application to Azure with continuous delivery

Solution 'ArtGalleryCRM' (6 projects)
ArtGalleryCRM (Portable)
ArtGalleryCRM.Droid
ArtGalleryCRM.iOS
ArtGalleryCRM.Uwp (Universal Windows)
ArtGalleryCRMService
ArtGalleryCRMSupportBot

The Agents have been started successfully

With the backend published and running, we can now move on to the frontend implementation with Telerik UI for Xamarin controls.

Building the UI with Telerik UI for Xamarin

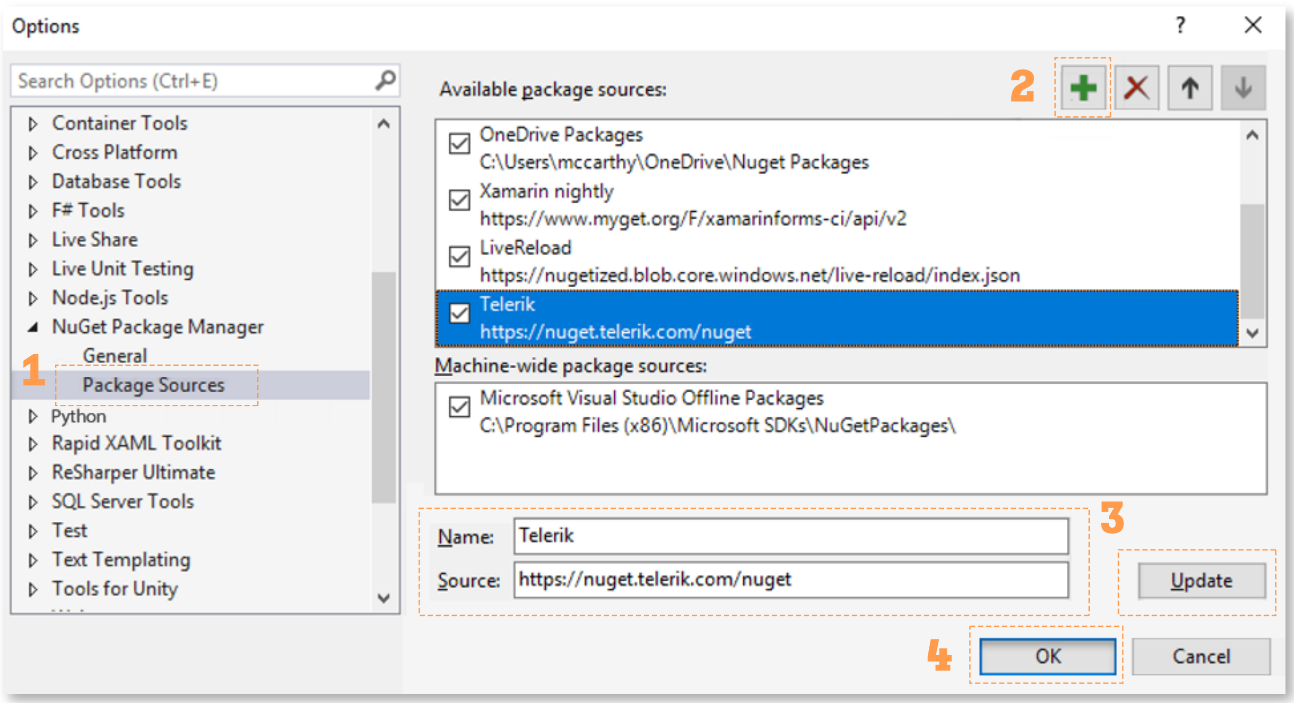
Before we get started building our beautiful UI, we'll want to add the assembly references and NuGet packages for the tools and components we'll want to use. Let's start with Telerik UI for Xamarin.

Installing Telerik UI for Xamarin

There are two primary routes to add Telerik UI for Xamarin to the projects: NuGet packages or locally installed assemblies. We used the NuGet package for this demo, but let's look at both options:

Option 1: Telerik NuGet Server

The Telerik NuGet packages are available from the Telerik NuGet Server. You will need to add this server to your Package Sources list. We have easy-to-follow instructions in the Telerik NuGet Server documentation. Here's a screenshot of how to add a source in Visual Studio:



Available package sources:

OneDrive Packages
C:\Users\mccarthy\OneDrive\Nuget Packages
Xamarin nightly
https://www.myget.org/F/xamarinforms-ci/api/v2
LiveReload
https://nugetized.blob.core.windows.net/live-reload/index.json
Telerik
https://nuget.telerik.com/nuget

Machine-wide package sources:

Microsoft Visual Studio Offline Packages
C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\

Name: Telerik
Source: https://nuget.telerik.com/nuget

Once you've added the server to your Package Sources list, you can right-click on the solution and select "Manage NuGet Packages for Solution". This will let you install a package to multiple projects at the same time. When the NuGet Package Manager appears, do the following:

1. Select "Telerik" as the Package source
2. Filter by "**Telerik.UI.for.Xamarin**"
3. Select the Xamarin.Forms projects
4. Click "Install"

You are now ready to start using the Telerik UI for Xamarin controls!

We recommend installing the NuGet package. It is faster to get started and, thanks to the NuGet Package Manager dependency resolution, it will install the other required dependencies for Telerik UI for Xamarin (e.g. Xamarin SkiaSharp packages and required Xamarin Android Support Library packages).

Option 2: Installed Assembly References

If you choose the assembly reference route, you can find the assemblies for each project in conveniently named folders in the Telerik UI for Xamarin installation folder. The default path is

C:\Program Files (x86)\Progress\Telerik UI for Xamarin [version]\Binaries

Within the Binaries folder, you will find subfolders corresponding to the project types. Each of these folders contains the assemblies that should be added to the project.

Pick and Choose Assembly References

If you only use one or two controls and don't plan on using the Linker to remove unused assemblies from the compiled app package, you can pick and choose individual assemblies from those folders.

To know what the required assemblies are for each control, review the control's "**Required Telerik Assemblies**" article. You will find that every control has this article underneath the "Getting Started" folder. As an example, here's the location of the RadListView's Required Telerik Assemblies article.

We recommend adding them all from the start so that you can be up and running quickly. After the project is done, you can go back and remove unused assemblies if necessary.

As mentioned above in Option 1, there are a couple of dependencies. If you choose the assembly reference route, you'll need to explicitly add these NuGet packages. Let's go over them one by one.

SkiaSharp Dependency

Several of the Telerik UI for Xamarin controls have a dependency on Xamarin's SkiaSharp libraries. Add the following packages to the projects:

- SkiaSharp (v 1.68.0 to all projects)
- SkiaSharp.Views (v 1.68.0 to all projects **except class library**)
- SkiaSharp.Views.Forms (v 1.68.0 to all projects)

Xamarin Android Support Libraries

The native Android controls need a few Xamarin Android Support Libraries to operate properly. Go to the Required Android Support Libraries article to see the full list. You only need to install these to the Android project.

Azure Mobile Service Client SDK

One of the main reasons we started with the download starter app is because the Microsoft Azure Mobile Client SDK NuGet package is already installed and has a TodoItemManager class that uses the SDK to interact with the backend.

Open the class library project's Constants.cs file and notice that the service URL will already be set in the ApplicationURL property, like this:

```
public static class Constants

{

    // Replace strings with your Azure Mobile App endpoint.

    public static string ApplicationURL = "https://YOUR_URL.azurewebsites.
net";

}
```

Data Models

Notice the TodoItem.cs data model that comes with the project. Before deleting it, you can use it as a template for the four data model classes you need to interact with the backend:

- Customer.cs
- Employee.cs
- Order.cs
- Product.cs

These classes should have the same properties as the entity models in the server application, but with some extra properties defined in a shared base class:

```
public class BaseDataObject : ObservableObject
{   
    public BaseDataObject()

    {

        Id = Guid.NewGuid().ToString();

    }

    [JsonProperty(PropertyName = "id")]

    [Ignore]

    public string Id { get; set; }
```

```
        [JsonProperty(PropertyName = “createdAt”)]

        [Ignore]

        public DateTimeOffset CreatedAt { get; set; }

        [UpdatedAt]

        [Ignore]

        public DateTimeOffset UpdatedAt { get; set; }

        [Version]

        [Ignore]

        public string AzureVersion { get; set; }

    }
```

Service Classes

With the models defined, we now turn our attention to the TodoItemManager class. This is the workhorse that syncs data between Azure and the app, but also manages the offline database syncing if you enabled it.

For the Art Gallery CRM app, we need four manager classes. Instead of having multiple methods or complicated overloads, we created an interface that defines those CRUD operations:

```
public interface IDataStore<T>

{

    Task<bool> AddItemAsync(T item);

    Task<bool> UpdateItemAsync(T item);

    Task<bool> DeleteItemAsync(T item);

    Task<bool> PurgeAsync();

    Task<T> GetItemAsync(string id);

    Task<string> GetIdAsync(string name);

    Task<IReadOnlyList<T>> GetItemsAsync(bool forceRefresh = false);

    Task<bool> PullLatestAsync();

    Task<bool> SyncAsync();

}
```

We also need a way to share a single instance of the MobileServiceClient, otherwise we would need to have separate instances of the client in each service class, which would unnecessarily increase the app's memory footprint.

One option we can use is to place the MobileServiceClient reference in the App class to make the same instance available to all the services.

With the interface defined and the MobileServiceClient reference available globally, we can start implementing a DataStore service class for each entity model.

As an example, here is the Customer entity's DataStore. In the constructor, we just set the reference for the CustomersTable and the interface-required methods can interact with that reference.

The example above shows how straightforward it is to use the MobileServiceClient to interact with the backend service. The SDK will make the call to the controller and return the items as a strongly typed list of Customer items.

Views and ViewModels

With the data layer ready, we can now start working on the UI. The starter app has a single page with a ListView that was populated with a list of to-do items using a code-behind approach. We need a more robust way to move around the app, as well as to view and edit data.

You can use a tabbed style or a master-detail style page layout. We decided to go with master-detail UI as it makes more sense to have top-level pages in the MasterPage (the side pane) and the drill-down pages (details and edit) in the DetailPage.

The top-level pages we want are:

- Employees
- Customers
- Products
- Orders
- Shipping
- Help
- About

Here, we use only the Employees table to demonstrate and implement concepts, views and view models.

To learn more about the general Xamarin.Forms concepts, you can consult the tutorial in the Xamarin documentation. The master-detail scenario is one example of such a concept. You can find the setup in this Xamarin.Forms Master-Details tutorial.

From this point on, assume the master-detail infrastructure is in place and we're adding pages to that setup. You can also reference the demo's source code on GitHub to follow along.

We used a naming convention where the list, detail and edit view models use the same name as the pages. This makes it easy to follow the responsibility of the view models:

- **Employees**Page & **Employees**ViewModel
- **EmployeeDetails**Page & **EmployeeDetail**ViewModel
- **EmployeeEdit**Page & **EmployeeEdit**ViewModel

EmployeesViewModel

When we start with the view model, we can ensure we have the properties we need to populate the view. This is rather straightforward—a single ObservableCollection to hold the items and a Task to load the data into the collection:

```
public class EmployeesViewModel : PageViewModelBase

    {
        private IReadOnlyList<Employee> _allEmployees;
        private string _searchText = string.Empty;
        public EmployeesViewModel()
        {
            this.Title = “Employees”;
            this.ItemTapCommand = new Command<ItemTapCommandContext>(this.
ItemTapped);
            this.ToolbarCommand = new Command(this.ToolbarItemTapped);
        }
        public ObservableCollection<Employee> Employees { get; } = new Obser
vableCollection<Employee>();
        public string SearchText
        {
            get => _searchText;
            set
```

```
                {
                    SetProperty(ref _searchText, value);
                    this.ApplyFilter(value);
                }
            }
            public ICommand ItemTapCommand { get; set; }
            public ICommand ToolbarCommand { get; set; }
            public override async void OnAppearing()
            {
                await this.LoadEmployeesAsync();
            }
            private void ApplyFilter(string textToSearch)
            {
                if (this._allEmployees == null)
                {
                    return;
                }
                var filteredEmployees = string.IsNullOrEmpty(textToSearch)
                    ? this._allEmployees
                    : this._allEmployees.Where(p => p.Name.ToLower().
    Contains(textToSearch.ToLower()));
                this.Employees.Clear();
                foreach (var employee in filteredEmployees)
                {
                    this.Employees.Add(employee);
                 }
            }
            private async Task LoadEmployeesAsync()
            {
                if (this.IsBusy)
                    return;
                try
                {
                    if (this.Employees.Count == 0)
                    {
                        this.IsBusy = true;
                        this.IsBusyMessage = "loading employees...";
                        this._allEmployees = await DependencyService.
    Get<IDataStore<Employee>>().GetItemsAsync();
                        foreach (var employee in this._allEmployees)
                        {
                            this.Employees.Add(employee);
                        }
                    }
                }
                catch (Exception ex)
                {
                    MessagingCenter.Send(new MessagingCenterAlert
                    {
                        Title = “Error”,
                        Message = $”There was a problem loading employees, check 
    your network connection and try again. Details: \r\n\n{ex.Message}”,
                        Cancel = “OK”
                    }, “Alert”);
                }
                finally
                {
                    this.IsBusyMessage = “”;
                    this.IsBusy = false;
                }
            }
            private async void ItemTapped(ItemTapCommandContext context)
            {
                if (context.Item is Employee employee)
                {
                    await this.NavigateForwardAsync(new EmployeeDetailPage(new Em
    ployeeDetailViewModel(employee)));
                }
            }
            private async void ToolbarItemTapped(object obj)
            {
                switch (obj.ToString())
                {
                    case “sync”:
                        this.Employees.Clear();
                        await this.LoadEmployeesAsync();
                        break;
                        case “add”:
                        await this.NavigateForwardAsync(new EmployeeEditPage());
                        break;
                }
            }
        }
```

Let’s look at how the items are retrieved from Azure:

```
await DependencyService.Get<IDataStore<Employee>>().
GetItemsAsync(forceRefresh);
```

This will call the method on the DataStore service class for the Employee entity. All the heavy lifting is neatly packed away either in the SDK itself or in the service class and keeps the ViewModel very readable.

EmployeesPage

With the ViewModel ready, we can move on to the UI and our first Telerik UI for Xamarin controls. We want not only to be able to list the employees, but also quickly filter them. We can do this with a combination of UI for Xamarin controls:

- RadAutoComplete to filter the employees
- RadListView to display the filtered results
- RadBusylindicator to show a busy overlay while data operations are occurring

RadListView and RadEntry

With the BindingContext of the page set to an instance of the EmployeesViewModel, we have an Employees property that will contain the list returned from the backend. We will use that as the ItemsSource of the RadListView:

```
<dataControls:RadListView ItemsSource=”{Binding Employees}”
                                  Style=”{StaticResource BaseListViewStyle}”
                                  Grid.Row=”1”>
```

Now we need to create an ItemTemplate for the RadListView to display the Employee properties we need, the name and the photo. The BindingContext of the ItemTemplate is the individual employee object in the FilteredItems collection. Therefore, we can bind a Label’s Text property to “Name” and bind an Image control’s Source property to “PhotoUri”.

Now we can move on to the filtering using RadEntry and its Text property:

```
<input:RadEntry x:Name="FilterEntry"

                Text="{Binding SearchText, Mode=TwoWay}"

                WatermarkText="Search"

                Margin="20,10,10,0">
```

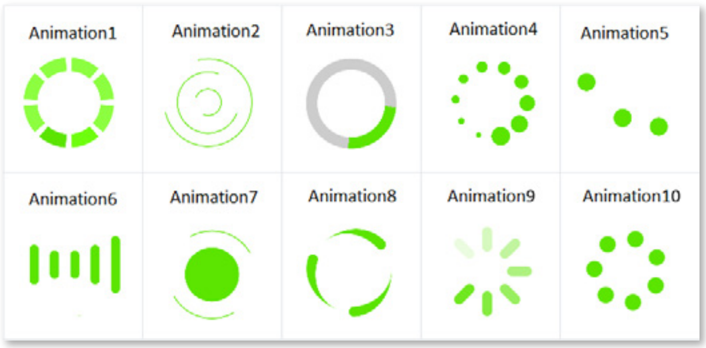
The Text is bound to the SearchText property from the EmployeesViewModel, where the filtering logic is implemented using the ApplyFilter method. This method executes custom filtering and then updates the Employees collection so that only filtered items are present and visualized in the RadListView:

```
public string SearchText

  {

      get => _searchText;
      set

      {   SetProperty(ref _searchText, value);
          this.ApplyFilter(value);

      }
  }
```

RadBusyIndicator

Now let's look at the **RadBusyIndicator**. This control offers a nice set of custom animations out of the box. You can also create your own custom animations if you need to. Since this app is designed for a professional audience, we used **Animation6**. See the [Animations documentation article]() for more details.

If you look at the ViewModel code, you will see that we toggle the BaseViewModel's IsBusy property when loading data:



Note: There are two ways to use the BusyIndicator—as an overlay, as shown in this demo, or through the control's content property. In this scenario, we want to use the overlay option since the BusyIndicator's content shadows the visual tree, which is undesirable with data-bound controls.

Finally, we want the user to be able to select an item in the RadListView and navigate to the EmployeeDetailsPage. We accomplish this by using a ListViewUserCommand for ItemTap.

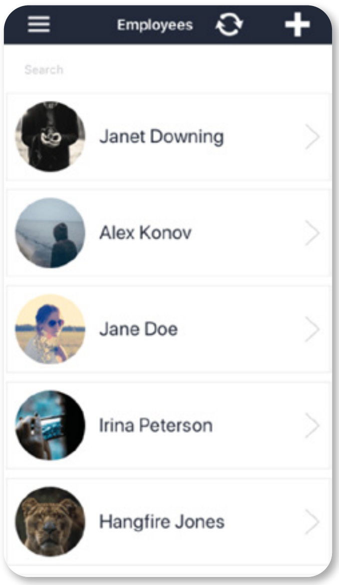
```
<dataControls:RadListView ItemsSource="{Binding Employees}"
                  Style="{StaticResource BaseListViewStyle}"
                  Grid.Row="1">
            <dataControls:RadListView.Commands>
                <commands:ListViewUserCommand Id="ItemTap"
                           Command="{Binding ItemTapCommand}" />
            </dataControls:RadListView.Commands>
<dataControls:RadListView>
```

The EmployeesViewModel's ItemTapped action is passed ItemTapCommandContext, which allows us to pass the selected employee to a new instance of EmployeeDetailPage.

```
private async void ItemTapped(ItemTapCommandContext context)
{
    if (context.Item is Employee employee)
    {
        await this.NavigateForwardAsync(new EmployeeDetailPage(new EmployeeD
etailViewModel(employee)));
    }
}
```

Take notice that we are passing the selected employee object to the constructor of the EmployeeDetailsViewModel.

Here's what the result looks like at runtime on iOS:



EmployeeDetailViewModel

When the ViewModel is constructed, we have the value of the employee passed in as a constructor parameter and we can set the SelectedEmployee right away:

```
public class EmployeeDetailViewModel : PageViewModelBase

{

    public EmployeeDetailViewModel(Employee item = null)

    {

        this.SelectedEmployee = item;

        ...

    }

    public Employee SelectedEmployee

    {

        get => this._selectedEmployee;

        set => SetProperty(ref this._selectedEmployee, value);

    }

    ...

}
```

We also want to generate data about this employee to populate charts, gauges and other nice UI features. To support this, we've added collections to hold chart data points and other data, along with a Task to load and calculate the data.

Let's take a closer look at **LoadDataAsync.** This is where we do the calculations that result in the values used for the view's chart and gauges.

There are two collections, one for compensation and one for sales revenue. We can calculate the compensation using the SelectedEmployee's values:

```
var bonusPercentage = (double)rand.Next(10, 20) / 100;

var benefitsPercentage = (double)rand.Next(5, 15) / 100;

var baseSalaryPercentage = 1 - bonusPercentage - benefitsPercentage;
```

```
CompensationData.Add(new ChartDataPoint {Title = “Base Salary”, Value =
SelectedEmployee.Salary * baseSalaryPercentage});

CompensationData.Add(new ChartDataPoint {Title = “Benefits”, Value =
SelectedEmployee.Salary * benefitsPercentage});

CompensationData.Add(new ChartDataPoint {Title = “Bonus”, Value =
SelectedEmployee.Salary * bonusPercentage});
```

For the sales revenue, we need to pull items from the Orders table and find matches for an employee’s ID:

```
// Gets all company orders

var orders = await DependencyService.Get<IDataStore<Order>>().
GetItemsAsync(true);

// Set company values

CompanySalesCount = orders.Count;

CompanySalesRevenue = Math.Floor(orders.Sum(o => o.TotalPrice));


// Get the orders associated with the employee

var employeeSales = orders.Where(o => o.EmployeeId == SelectedEmployee.Id).
OrderBy(o => o.OrderDate).ToList();


// Set employee values

EmployeeSalesCount = employeeSales.Count;

EmployeeSalesRevenue = Math.Floor(employeeSales.Sum(o => o.TotalPrice));



// Create Sales History chart data

foreach (var order in employeeSales)

{

    SalesHistory.Add(new ChartDataPoint

    {

        Value = order.TotalPrice,

        Date = order.OrderDate

    });

}
```

EmployeeDetailsPage

For the details page, we wanted a “hero” style header that holds the employee’s photo and other details like name and office location, while the body can contain the charts and gauges.

Here’s a high-level overview of this layout:

```
<Grid x:Name=”RootGrid”
              RowDefinitions=”{StaticResource AutoStarRowDefinitions}”>
            <StackLayout Style=”{StaticResource PageHeaderStackLayoutStyle}”>
               <ff:CachedImage Source=”{Binding SelectedEmployee.PhotoUri}”…>
               <StackLayout Padding=”10”>
                    <Label Text=”{Binding SelectedEmployee.Name}”… />
                    <Label Text=”{Binding SelectedEmployee.OfficeLocation}”…/>
               </StackLayout>
            </StackLayout>

            <Grid x:Name=”MetricsGrid”
                  BackgroundColor=”{StaticResource LightBackgroundColor}”
                  Grid.Row=”1”>
                <Grid.RowDefinitions … >
                <Grid.ColumnDefinitions … >

                <!-- SEPARATOR -->
                <BoxView BackgroundColor=”{StaticResource MediumBaseColor}”…>

                <!-- TOP RIGHT - Vacation Radial Gauge -->
                <Grid Grid.Row=”0”…>

                <!-- TOP LEFT - Salary Pie Chart -->
                <Grid x:Name=”PieChartGrid” … >

                <!-- SEPARATOR -->
                <BoxView BackgroundColor=”{StaticResource SeparatorColor}”…>

                <!-- BOTTOM LEFT - Sales Radial Gauge -->
                <Grid Grid.Row=”2” … >

                <!-- BOTTOM RIGHT - Revenue Linear Gauge -->
                <Grid Grid.Row=”2” … >
```

```
                    <!-- SEPARATOR -->

                    <BoxView BackgroundColor="{StaticResource SeparatorColor}"… >

                    <!-- BOTTOM - Sales History Cartesian Chart (Bar Series) -->
                    <Grid x:Name="HistoryChartGrid" … >

                <primitives:RadBusyIndicator x:Name="BusyIndicator" … >
```

When the page's OnAppearing method is called, the ViewModel's **LoadDataAsync** is called because we need to use asynchronous code. This makes the app much more responsive and snappier compared to doing the calculations from the page, the ViewModel constructor or using .Wait() on asynchronous methods.

Note: You should avoid using .Wait as much as possible.

Here's the OnAppearing method:

```
protected override async void OnAppearing()

{
    base.OnAppearing();
    await vm.LoadDataAsync();
    SetupVacationGauge();
    SetupSalesGauge();
    SetupRevenueGauge();
    SetupChartLegend();
}
```

You will notice that we configure some UI elements here as well. The chart series will automatically update when the bound ObservableCollections are populated, but the gauges use a little more fine-grained control.

Charts

Starting with the SalesHistory and Compensation charts, the series ItemsSources can be bound directly to the respective ViewModel collections:

- LineChart
- PieChart
- Gauges

The gauges are defined in the XAML, but configured in the clearly named methods in

OnAppearing. The Vacation RadialGauge is one example of such setup.

Here is the configuration method:

```
private void SetupVacationGauge()

{
    // Vacation Radial Gauge

    VacationGauge.StartAngle = 90;
    VacationGauge.SweepAngle = 360;
    VacationGauge.AxisRadiusFactor = 1;
    VacationLinearAxis.Maximum = vm.SelectedEmployee.VacationBalance;
    VacationLinearAxis.StrokeThickness = 0;
    VacationLinearAxis.ShowLabels = false;
    VacationRangesDefinition.StartThickness = 10;
    VacationRangesDefinition.EndThickness = 10;
    VacationRange.To = vm.SelectedEmployee.VacationBalance;
    VacationRange.Color = (Color)Application.Current.Resources
[“GrayBackgroundColor”];
    VacationIndicator.Value = vm.SelectedEmployee.VacationUsed;
    VacationIndicator.Fill = (Color)Application.Current.
Resources[“AccentColor”];
    VacationIndicator.StartThickness = 10;
    VacationIndicator.EndThickness = 10;
    VacationIndicator.StartCap = GaugeBarIndicatorCap.ConcaveOval;
    VacationIndicator.EndCap = GaugeBarIndicatorCap.Oval;

}
```

The same pattern is used for the other gauges as well.

Deleting an Employee

We also enable the user to delete the item through the EmployeeDetailPage. This is done using a Delete toolbar button Command.

Still, deleting an employee will also delete any records in the Orders table that contain orders with this employee’s ID. We need to first check if they are associated with any open orders and warn the user that deleting the employee will also delete the orders.

Here is the command’s Action in the EmployeeDetailViewModel:

```
private async Task DeleteEmployeeAsync()
{
    if (this.SelectedEmployee == null)
    {
        return;
    }

    var result = await App.Current.MainPage.DisplayAlert("Delete?", "Do you
really want to delete this item?", "Yes", "No");

    if (!result)
    {
        return;
    }

    try
    {
        this.IsBusy = true;
        // Do Referential Integrity Check
        var orders = await DependencyService.Get<IDataStore<Order>>().
GetItemsAsync();
        var matchingOrders = orders.Where(o => o.EmployeeId == this.
SelectedEmployee.Id).ToList();

        if (matchingOrders.Count > 0)
        {
            var deleteAll = await App.Current.MainPage.DisplayAlert("!!!
WARNING !!!", $"There are orders in the database for {_vm?.SelectedEmployee.
Name}. If you delete this employee, you'll also delete all of the associated
orders.\r\n\nDo you wish to delete everything?", "Delete All", "Cancel");

            // Back out if user declines to delete associated orders

            if (!deleteAll)
                return;
            // Delete each associated Order
            foreach (var order in matchingOrders)
            {
                await DependencyService.Get<IDataStore<Order>>().
DeleteItemAsync(order);
            }
        }

        // Lastly, delete the employee
        await DependencyService.Get<IDataStore<Employee>>().
DeleteItemAsync(this.SelectedEmployee);
    }
    catch (Exception ex)
    {
        // display error to user
    }

    finally
    {
        this.IsBusy = false;
    }

     await this.NavigateBackAsync();
}
```

Note: There are other ways to handle this scenario. Deleting the related orders is a rather drastic measure. In a real-world app, you are more likely to reset the ID or use a nullable field for EmployeeId.

Editing an Employee

A common approach to editing is to build the edit form into the details page. We wanted to facilitate adding an employee with the same edit form.

We thought that a separate EmployeeEditPage would be better since it would allow us to make the page code more readable and keep responsibilities separate.

The same EmployeeDetailsPage toolbar that has the delete button also has buttons for Edit and New Order. They can all share the same view model Command, but invoke different operations depending on a predefined command parameter.

```
<ContentPage.ToolbarItems>
    <ToolbarItem Text="order"
                 IconImageSource="add_order.png"
                 Command="{Binding ToolbarCommand}"
                 CommandParameter="order" />
    <ToolbarItem Text="edit"
                 IconImageSource="edit.png"
                 Command="{Binding ToolbarCommand}"
                 CommandParameter="edit" />
    <<ToolbarItem Text="delete"
                 Icon="delete.png"
                 Command="{Binding ToolbarCommand}"
                 CommandParameter="delete" />
</ContentPage.ToolbarItems>
```

Here's the view model command **ToolbarItemTapped Action** that will navigate to the **EmployeeEditPage**, the **OrderEditPage** or delete. We pass the **SelectedEmployee** to the constructor of both options.

```
private async void ToolbarItemTapped(object obj)
{
    if (this.SelectedEmployee == null)
    {
        return;
    }

    switch (obj.ToString())
    {
       case "order":
           await this.NavigateForwardAsync(new OrderEditPage(this.
SelectedEmployee));
            break;
        case “edit”:
            await this.NavigateForwardAsync(new EmployeeEditPage(this.
SelectedEmployee));
            break;
        case “delete”:
            await this.DeleteEmployeeAsync();
            break;
    }
}
```

EmployeeEditViewModel

The EmployeeEditViewMode’s responsibility is simple:

- Determine whether this is a new employee or an existing employee
- Provide a mechanism to either insert a new or update an existing database record

To do this, use the following items in the EmployeeEditViewModel:

- SelectedEmployee property
- IsNewEmployee property
- UpdateDatabaseAsync method

If you look at a shortened version of the UpdateDatabaseAsync method, you will notice that, depending on the value of **IsNewEmployee**, we can choose to either update or insert SelectedEmployee:

```
public async Task UpdateDatabaseAsync()

{
    if (IsNewEmployee)

    {
        await DependencyService.Get<IDataStore<Employee>>().
AddItemAsync(SelectedEmployee);

    }

    else

    {
        await DependencyService.Get<IDataStore<Employee>>().
UpdateItemAsync(SelectedEmployee);

    }
}
```

Note: Some code is omitted for clarity. Please see the demo source for the full method.

Using the page constructor, we can determine whether this is a new or existing employee.

EmployeeEditPage

EditPage's XAML is very simple:

- An **Image + Button** to select a photo (instead of manually typing in a URL string into the DataForm)
- A RadDataForm to edit the fields

Here is the high-level layout:

```
<Grid x:Name="RootGrid"
             RowDefinitions="{StaticResource AutoStarRowDefinitions}">
            <StackLayout Style="{StaticResource PageHeaderStackLayoutStyle}">
               <ff:CachedImage Source="{Binding SelectedEmployee.PhotoUri}"…>
               <Button Text="Set Photo (random)"...>
             </StackLayout>
             <input:RedDataForm x:Name="DataForm"
               Source="{Binding SelectedEmployee}"
               FormValidationCompleted="DataForm_OnFormValidationCompleted"
               Grid.ROw="1"
               Margin="10,0,10,0">
                <input:RedDataForm.EditorStyle ... >
              </input:RedDataForm>
</Grid>
```

As you can see, the DataForm's Source is bound to the ViewModel's SelectedEmployee. The button's click handler just assigns a random photo to the SelectedEmployee's PhotoUri.

The extra interesting part of this page lies in the Employee model's property attributes that tell the RadDataForm how to create its Editors and the EditPage how to determine if we are editing or adding an employee.

Data Annotations

RadDataForm Data Annotations are model property attributes used to set a wide range of DataForm features. You can find a list of the attributes in the RadDataForm documentation under the "Data Annotations" subfolder.

For the Employee model, we need to exclude the **PhotoUri** property because we want the button to set that value instead of a user-entered string. To do this, we can leverage the Ignore attribute:

```
[Ignore]
public string PhotoUri

{
    get => photoUri;
    set => SetProperty(ref photoUri, value);
}
```

Data Annotations let you set the editor display and validation options as well. We also use the DisplayOptions, NonEmptyValidator and NumericalRangeValidator attributes on the Employee properties:

Page Constructor and Configuring Editors

The page needs to determine whether we are editing an employee or creating a new one. This is simply achieved by using two constructors.

When the overloaded constructor is used, we are passing an existing employee object to edit. When the normal constructor is used, we are creating a new employee:

```
// If we're adding a new employee

public EmployeeEditPage()

{
    InitializeComponent();
    ViewModel.IsNewEmployee = true;
    ViewModel.SelectedEmployee = new Employee();
    ConfigureEditors();
}


// If we›re editing an existing employee

public EmployeeEditPage(Employee employee)

{
    InitializeComponent();
    ViewModel.SelectedEmployee = employee;
    ConfigureEditors();

}
```

In the ConfigureEditors method, we set the EditorType for each property. This tells the DataForm to use a specific editor type. See the full list in the DataForm Editors documentation.

```
private void ConfigureEditors()
{   DataForm.RegisterEditor(nameof(Employee.Name), EditorType.TextEditor);
    DataForm.RegisterEditor(nameof(Employee.OfficeLocation),
EditorType.TextEditor);
    DataForm.RegisterEditor(nameof(Employee.HireDate), EditorType.DateEditor);
    DataForm.RegisterEditor(nameof(Employee.Salary), EditorType.DecimalEditor);
    DataForm.RegisterEditor(nameof(Employee.VacationBalance),
EditorType.IntegerEditor);
    DataForm.RegisterEditor(nameof(Employee.VacationUsed),
EditorType.IntegerEditor);
    DataForm.RegisterEditor(nameof(Employee.Notes), EditorType.TextEditor);
}
```

The final remaining item to cover is how the DataForm saves the data in its editors.

The DataForm has validation considerations, so the values in the editors are not immediately committed to the bound Source. Instead, you call **DataForm.CommitAll** and validation is performed against the values (according to the Data Annotations you have set).

The DataForm has a **FormValidationCompleted** event that will fire when values are committed to the form. This means we can use an **EventToCommandBehvaior** to hook up the event to a view model command.

```
<input:RadDataForm x:Name="DataForm"
                   Source="{Binding SelectedEmployee}"
                   Grid.Row="1"
                   Margin="10,0,10,0">
    <input:RadDataForm.Behaviors>
        <behaviors:EventToCommandBehavior EventName="FormValidationCompleted"
                     Command="{Binding FormValidationCompleteCommand}" />
    </input:RadDataForm.Behaviors>
</input:RadDataForm>
```

In the command, we use the event args to hold a flag on whether or not the SelectedEmployee's changed can be saved.

```
private void FormValidationCompleted(object obj)
{

if (obj is FormValidationCompletedEventArgs e)
    {
        this._isReadyToSave = e.IsValid;
    }
}
```

Just like the EmployeeDetailsPage, the EmployeeEditPage has a toolbar button that invokes a command. In the command's Action, we can save it and automatically navigate back to the details page when it is done.

```
private async void ToolbarItemTapped(object obj)
{    switch (obj.ToString())
    {   case "save":
            // Commit the Editor's values for validation
            this.DataFormView.CommitChanges();
            // If values pass validation
            if (this._isReadyToSave)
            {    // Save the record
                if (await this.UpdateDatabaseAsync())
                {  await this.NavigateBackAsync();
                }
            }
            break;
    }
}
```

Wrapping Up

This concludes our deep dive into the **ListPage -> DetailsPage -> EditPage** approach. The Art Gallery CRM app uses this pattern for all four of the entity models.

Now that you have a handle on the pattern, take a look at the following special scenarios and try explaining how the Art Gallery CRM app uses:

- **RadDataGrid** to easily group, sort and filter open orders on the OrdersPage
- **RadSlideView** for a great "out-of-the-box experience" on the WelcomePage
- **RadListView with GridLayout** to show beautiful product images on the ProductsPage
- **RadCalendar** to show collated shipping dates on the CalendarPage
- **RadNumericInput** and **RadMaskedInput** on the advanced OrderEditPage

Powering the Conversation

The Telerik UI for Xamarin RadChat control is intentionally designed to be platform agnostic. Although we use Azure for this project, you can use anything you want or currently have.

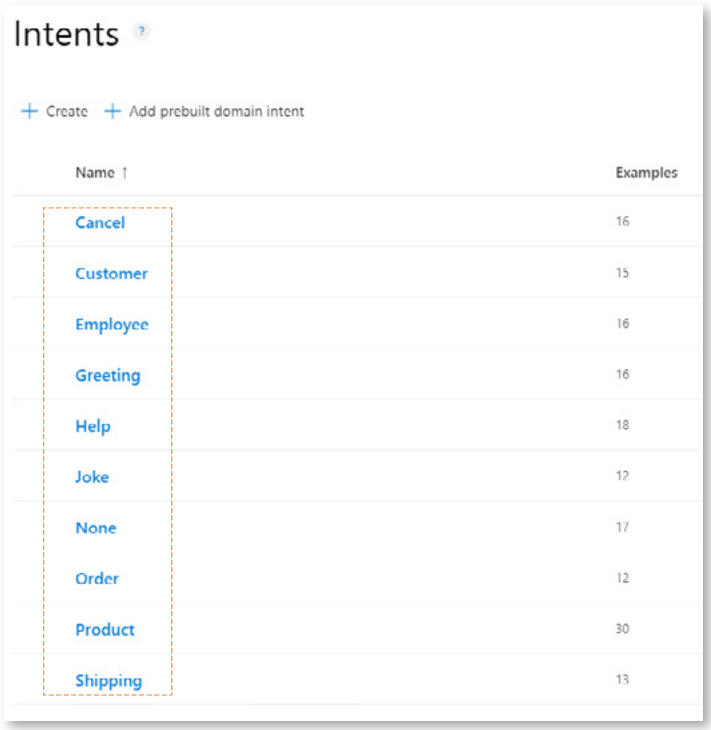
We wanted to ensure that the app had a dynamic and intelligent service communicating with the user, which is why we chose Azure. You can train a LUIS model to identify certain contexts and understand what the user is asking for. For example, it can discern whether the user is inquiring about product details or shipping information.

To get started, follow the Quickstart: create a new app in the LUIS portal tutorial. It's quick and easy to spin up a project and get started training the model with your custom intents.

At this point, you might want to think about whether you want to use Intents, Entities or both. The Art Gallery CRM support page is better served with Intents. To help you choose the right approach for you app, Microsoft has a nice comparison table linked here—Entity compared to Intent.

Intents

On the CRM app's chat page, the user is interacting with a support bot. We mainly need to determine if the user is asking for information about a product, employee, order or customer. To achieve this, we used the following Intents:

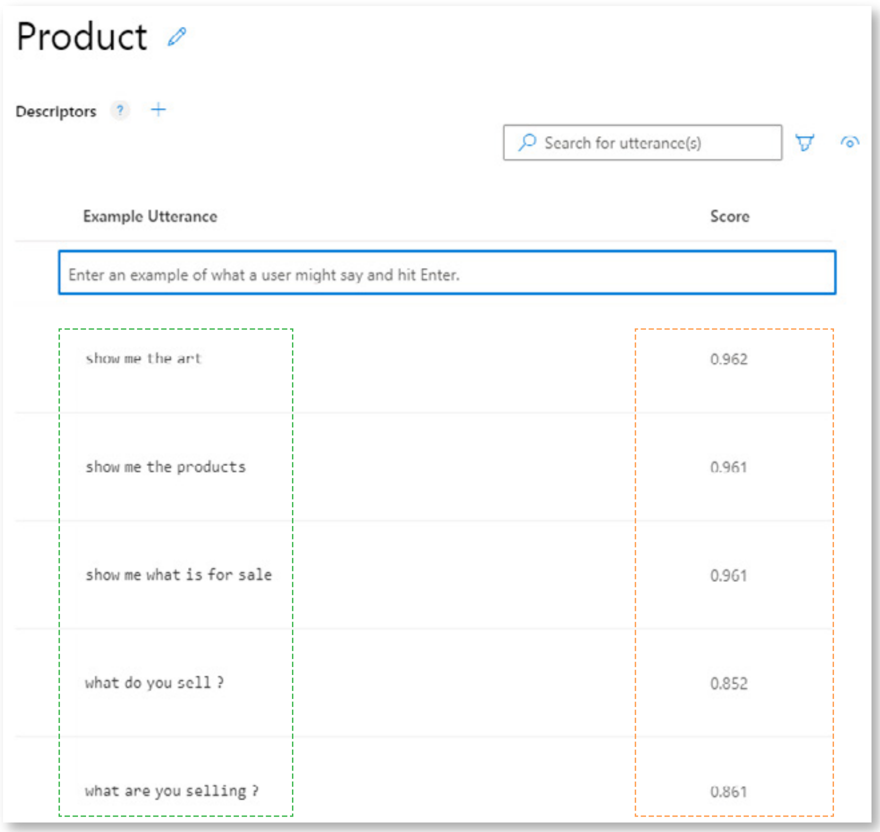


Intents

+ Create + Add prebuilt domain intent

Name ↑	Examples
Cancel	16
Customer	15
Employee	16
Greeting	16
Help	18
Joke	12
None	17
Order	12
Product	30
Shipping	13

To train the model to detect an intent, you give it a list of things the user might say. This is called an Utterance. Let's look at the utterances for the **Product** intent.


Product

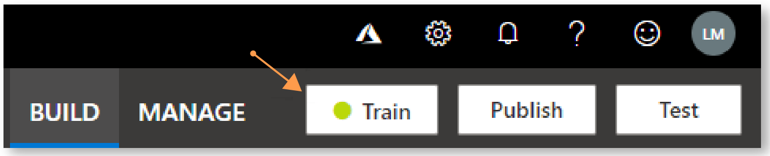
Descriptors

Example Utterance	Score
show me the art	0.962
show me the products	0.961
show me what is for sale	0.961
what do you sell ?	0.852
what are you selling ?	0.861

On the left side, you can see the example utterances we entered. These do not need to be full sentences, just key words or phrases you would expect in a user's question. On the right side, you see the current detection score that utterance has received after the last training.

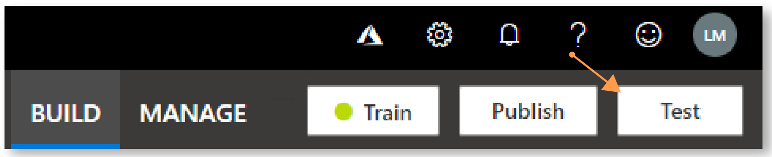
Training

You can start training the model by clicking the "Train" button in the menu bar at the top of the portal:

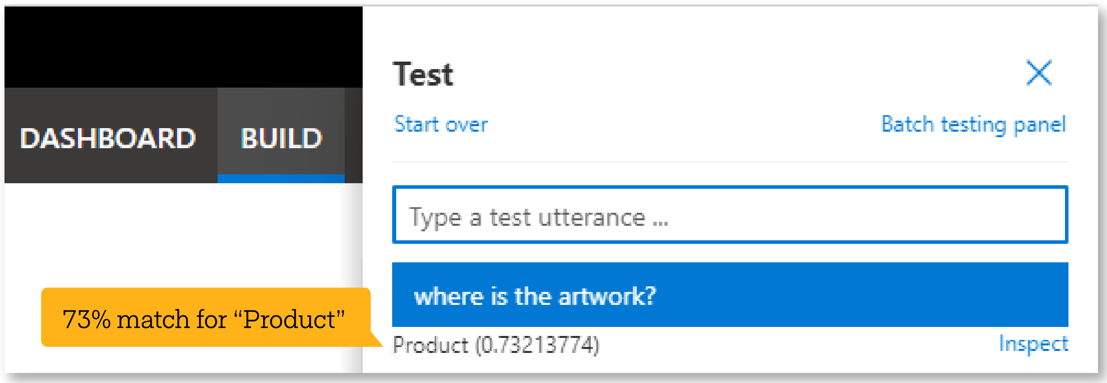


Testing

After LUIS is trained with the current set of Intents and Utterances, you can test how well it performs. Select the "Test" button:



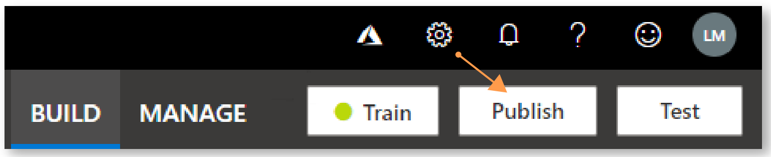
When you see a fly-out, enter your test question to fetch a result:



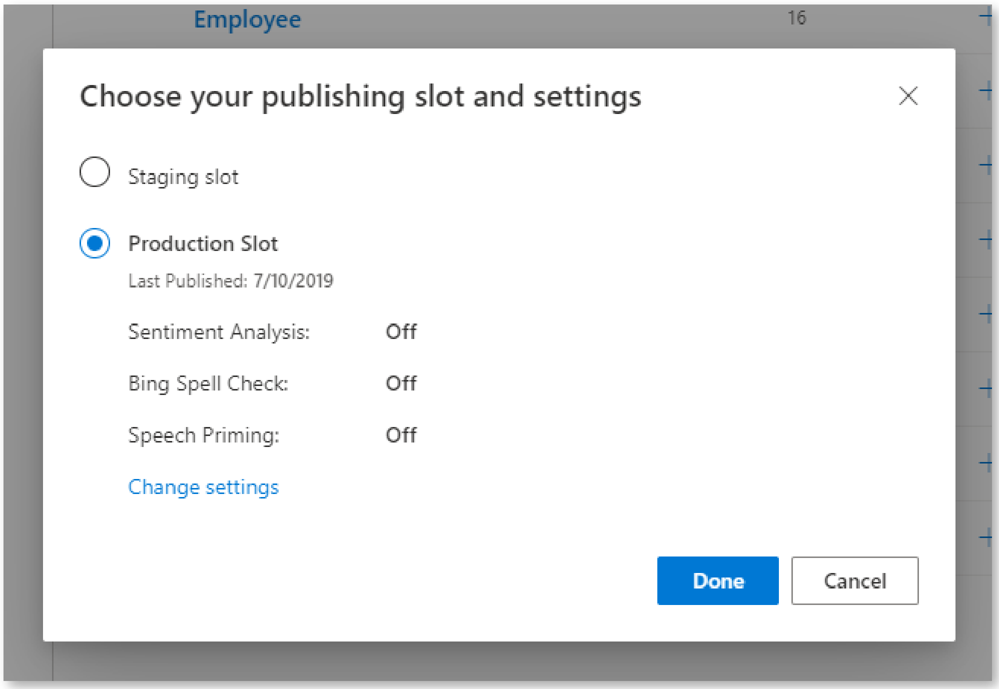
In the above test, we tried a search for "**where is the artwork?**" and the result was a 73% match for a **Product** intent. If you want to improve the score, add more utterances to the intent and retrain it.

Publishing

When you are satisfied with the scores you get for the Intents, you can publish the model. Use the "Publish" button:



A dialog will open, prompting you to pick a release slot to push the changes to. You can stage the new changes or push them to production:


Employee 16

Choose your publishing slot and settings

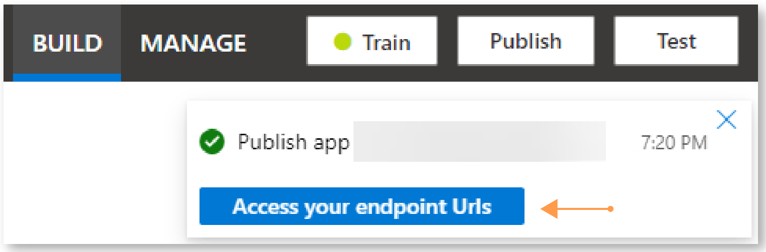
Staging slot [✓ Production Slot]

Last Published: 7/10/2019

Sentiment Analysis:	Off
Bing Spell Check:	Off
Speech Priming:	Off

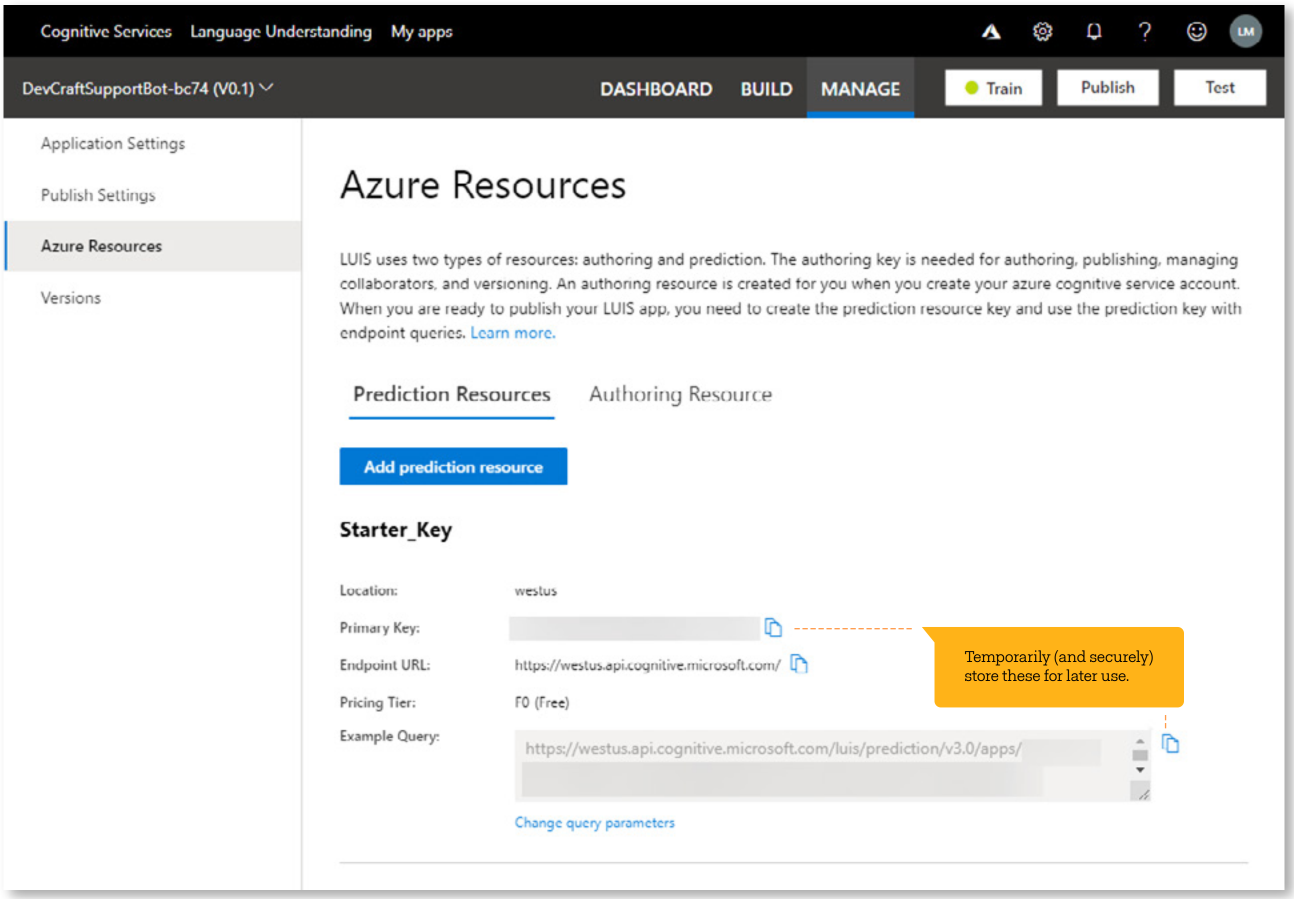
API Endpoints

When the publishing is complete, you will see a button in the notification to navigate to the Manage tab where you can see the REST URLs for your release:



Publish app 7:20 PM

In the Manage tab, the Azure Resources pane will be preselected. In the pane, you will see the API endpoints and other important information.



Cognitive Services Language Understanding My apps

DevCraftSupportBot-bc74 (V0.1)

Azure Resources

LUIS uses two types of resources: authoring and prediction. The authoring key is needed for authoring, publishing, managing collaborators, and versioning. An authoring resource is created for you when you create your azure cognitive service account. When you are ready to publish your LUIS app, you need to create the prediction resource key and use the prediction key with endpoint queries. Learn more.

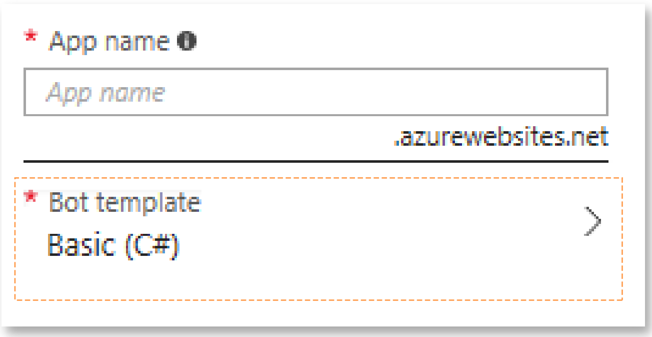
Starter_Key

Location:	westus
Primary Key:	
Endpoint URL:	https://westus.api.cognitive.microsoft.com/
Pricing Tier:	F0 (Free)
Example Query:	https://westus.api.cognitive.microsoft.com/luis/prediction/v3.0/apps/

Note: We will come back to these values later. You can temporarily store them in a secure location. Do not share your primary key or example query (it has the primary key in the query parameters).

Azure Bot Service

Follow [the Azure Bot Quickstart tutorial]on how to create a new Bot Service in your Azure portal. When you go through the configuration steps, be sure to choose the C# Bot template so that the bot's backend will be an ASP.NET application:



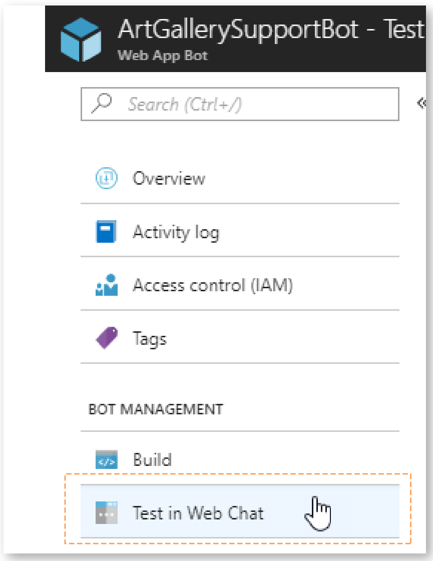
* App name

App name .azurewebsites.net

* Bot template
Basic (C#)

First Use: Testing with WebChat

To make sure things are running after you have published the project to Azure, open the bot's blade and select the "Test in Web Chat" option:



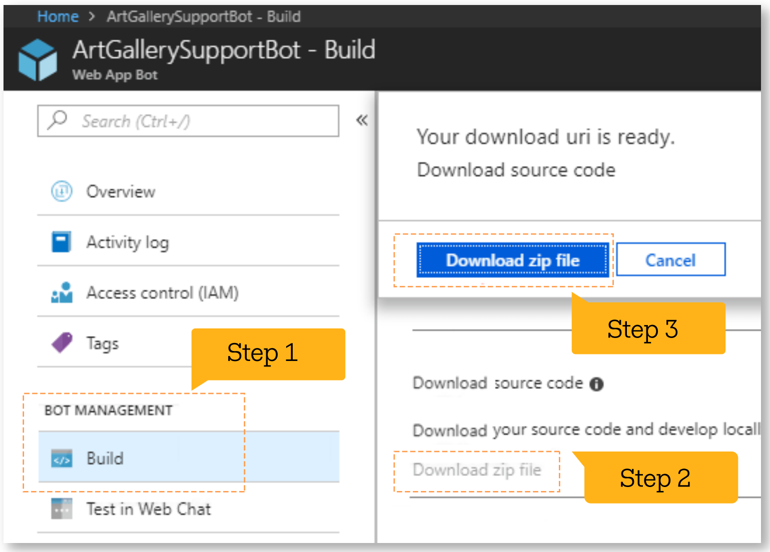
This will open a blade and initialize the bot. When it's ready, you can test the communication. The default bot template uses an "echo" dialog (covered later).

Test this by sending any message. You should see your message prefixed with a count number. Next, try sending the word "reset". You will get a dialog prompt to reset the message count.

Explore the Code

Once the bot is responding and working, let's look at the code by downloading a copy of the ASP.NET application:

- Step 1: Open the Build blade
- Step 2: Click the "Download zip file" link (this will generate the zip)
- Step 3: Click the "Download zip file" button (this will download the zip)



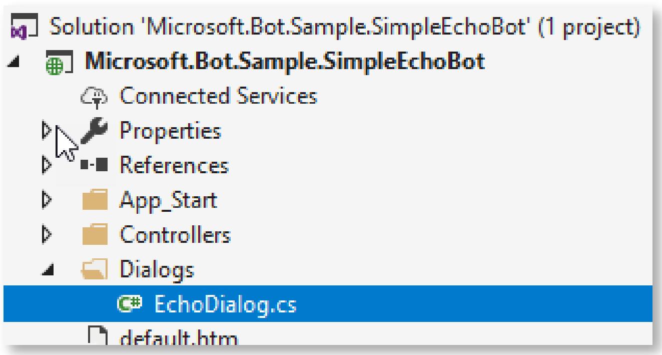
Look for a ZIP file named similarly "YourBot-src". You can now unzip and open the **Microsoft.Bot.Sample.SimpleEchoBot.sln** in Visual Studio. Do a Rebuild to restore the NuGet packages.

Review the workflow by opening the **MessagesController.cs** class in the **Controllers** folder and look at the POST method. The method has an Activity parameter, which is sent to the controller by the client application. We can check what type of activity this is and act on that.

```
[ResponseType(typeof(void))]

public virtual async Task<HttpResponseMessage> Post([FromBody] Activity
activity)

{

    // If the activity type is a Message, invoke the Support dialog

    if (activity != null && activity.GetActivityType() ==ActivityTypes.
Message)

    {
        await Conversation.SendAsync(activity, () => newDialogs.EchoDialog());

    }

...

}
```

In the case of our echo bot, this is **ActivityType.Message**, in which case the logic determines we respond with an instance of **EchoDialog**.

Now, let's look at the **EchoDialog** class, this is in the **Dialogs** folder:


Solution 'Microsoft.Bot.Sample.SimpleEchoBot' (1 project)
Microsoft.Bot.Sample.SimpleEchoBot
Connected Services
Properties
References
App_Start
Controllers
Dialogs
EchoDialog.cs
default.htm

There are a few methods in the class, but we are focusing on the **MessageReceivedAsync** method:

```
public async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<IMessageActivity> argument)

{    var message = await argument;

    if (message.Text == "reset")

    {

        // If the user sent "reset", then reply with a prompt dialog

        PromptDialog.Confirm(

            context,
            AfterResetAsync,
            "Are you sure you want to reset the count?",
            "Didn't get that!",
            promptStyle: PromptStyle.Auto);

    }

    else

    {
        // If the user sent anything else, reply back with
the same message prefixed with the message count number

        await context.PostAsync($"{this.count++}: You said {message.Text}");
        context.Wait(MessageReceivedAsync);

    }

}
```

This is the logic that the bot used during the initial test in Web Chat. Here you can see how all the messages you sent were echoed and why it was prefixed with a message count.

Xamarin.Forms Chat Service

To set up the client side of the chat service, you need a couple of things:

- A bot service class to send and receive messages
- A Xamarin.Forms page with a Telerik UI for Xamarin Conversational UI control

Bot Service Class and DirectLine API

In addition to the embeddable WebChat control, the Bot Service offers to directly interact with it using network requests via the DirectLine API.

Although you could manually construct and send HTTP requests, Microsoft has provided a .NET client SDK that makes it very simple and painless to connect to and interact with the Bot Service via the Microsoft.Bot.Connector.DirectLine NuGet package.

With this, we could technically just use it directly in the ViewModels (or code behind) of the Xamarin.Forms page. We created a service class that is not tightly bound to that specific page.

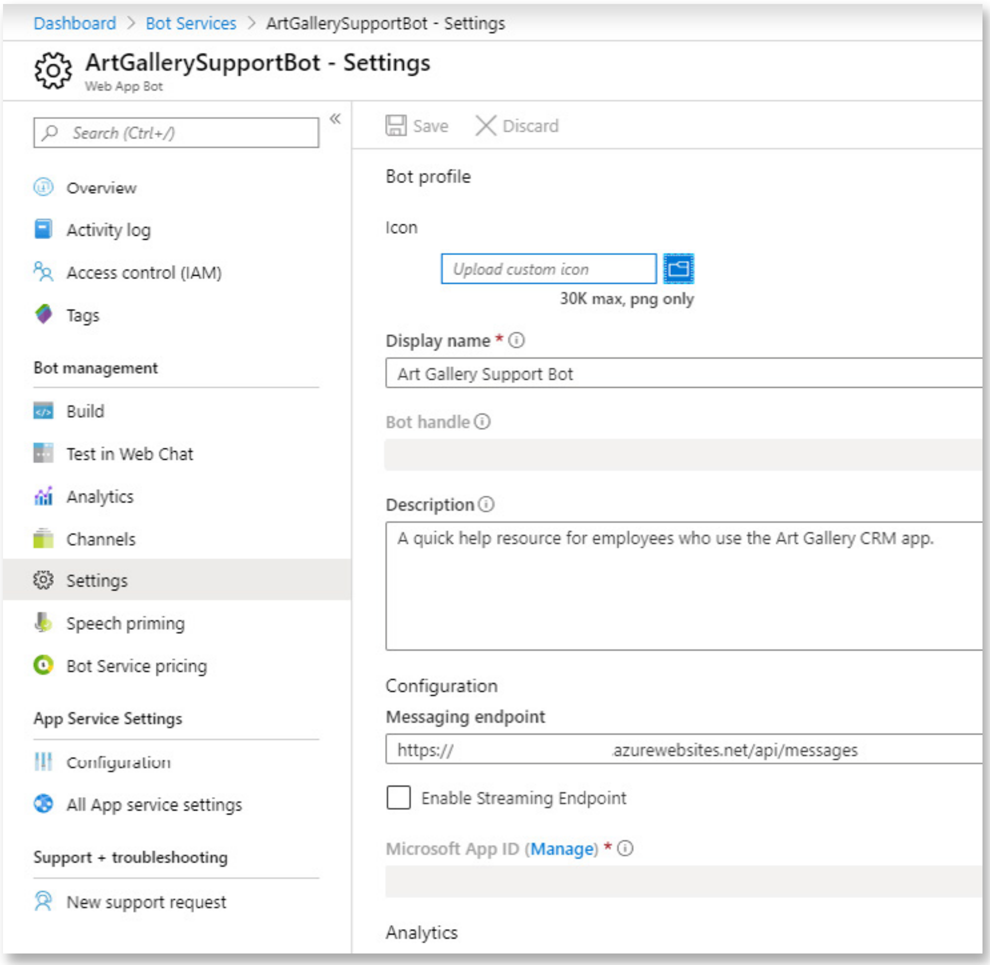
Here is the service class:

```
using System;
using System.Linq;
using System.Threading.Tasks;
using ArtGalleryCRM.Forms.Common;
using Microsoft.Bot.Connector.DirectLine;
using Xamarin.Forms;

namespace ArtGalleryCRM.Forms.Services
{
    public class ArtGallerySupportBotService
    {
        private readonly string _user;
        private Action<Activity> _onReceiveMessage;
        private readonly DirectLineClient _client;
        private Conversation _conversation;
        private string _watermark;
        public ArtGallerySupportBotService(string userDisplayName)
        {
```

```
// New up a DirectLineClient with your App Secret

            this._client = newDirectLineClient(ServiceConstants.
DirectLineSecret);

            this._user = userDisplayName;
        }

        internal void AttachOnReceiveMessage(Action<Activity>
onMessageReceived)

        {
            this._onReceiveMessage = onMessageReceived;
        }


        public async Task StartConversationAsync()
           {

            this._conversation = await _client.Conversations.
StartConversationAsync();

        }

// Sends the message to the bot

        public async void SendMessage(string text)

        {

            var userMessage = new Activity

            {
                From = new ChannelAccount(this._user),
                Text = text,
                Type = ActivityTypes.Message

            };

            awaitthis._client.Conversations.PostActivityAsync
(this._conversation.ConversationId, userMessage);


            await this.ReadBotMessagesAsync();

        }


// Shows the incoming message to the user

        private async Task ReadBotMessagesAsync()

        {
            var activitySet = awaitthis._client.Conversations.
GetActivitiesAsync(this._conversation.ConversationId, _watermark);

            if (activitySet != null)

            {
                this._watermark = activitySet.Watermark;

                var activities = activitySet.Activities.Where(x => x.From.Id
== ServiceConstants.BotId);

                Device.BeginInvokeOnMainThread(() =>

                {
                    foreach (Activity activity in activities)

                    {
                        this._onReceiveMessage?.Invoke(activity);
                    }
                });
            }
        }
    }
}
```

There are two responsibilities—send messages to the bot and show incoming bot messages to the user. In the class constructor, we instantiate a **DirectLineClient** object. Notice that it needs an **App ID** parameter. You can find this in the Azure Bot Service Settings page.

Here is a screenshot to guide you. The ID is the "Microsoft App ID" field:



Bot profile

Icon

Upload custom icon

30K max, png only

Display name *

Art Gallery Support Bot

Bot handle

Description

A quick help resource for employees who use the Art Gallery CRM app.

Configuration

Messaging endpoint

https://   azurewebsites.net/api/messages

☐ Enable Streaming Endpoint

Microsoft App ID (Manage) *

Analytics

ViewModel

To hold the conversation messages, we used an **ObservableCollection<TextMessage>**. The RadChat control supports MVVM scenarios. You can learn more about it in the Telerik UI RadChat MVVM Support documentation.

```
public ObservableCollection<TextMessage> ConversationItems { get; set; }
= new ObservableCollection<TextMessage>();
```

The service instance can be defined as a private field that is created when the ViewModel is initialized:

```
public class SupportViewModel : PageViewModelBase

{

    private ArtGallerySupportBotService botService;

    public async Task InitializeAsync()

    {

        this.botService = new ArtGallerySupportBotService(“Lance”);

        this.botService.AttachOnReceiveMessage(this.OnMessageReceived);

    }

...

}
```

When a message from the bot comes in, the service will invoke the ViewModel's OnMessageReceived method and pass an activity parameter. With this information, you can add a message to the ConversationItems collection:

```
private void OnMessageReceived(Activity activity)

{
    ConversationItems.Add(new TextMessage

    {
        Author = this.SupportBot, // the author of the message

        Text = activity.Text // the message

    });
}
```

For the user's outgoing message, we used the ConversationItems's **CollectionChanged** event. If the message that was added is from the user, then we can give it to the service class to be pushed to the bot.

```
private void ConversationItems_CollectionChanged(object sender,
NotifyCollectionChangedEventArgs e)

{

    if (e.Action == NotifyCollectionChangedAction.Add)

    {

        // Get the message that was just added to the collection

        var chatMessage = (TextMessage)e.NewItems[0];


        // Check the author of the message

        if (chatMessage.Author == this.Me)

        {

            // Debounce timer

            Device.StartTimer(TimeSpan.FromMilliseconds(500), () =>

            {

                Device.BeginInvokeOnMainThread(() =>

                {

                    // Send the user's question to the bot!

                    this.botService.SendMessage(chatMessage.Text);

                });


                return false;

            });

        }

    }

}
```

The XAML

In the view, all you need to do is bind the **ConversationItems** property to the RadChat's ItemsSource property. When the user enters a message in the chat area, a new TextMessage is created by the control and inserted into the ItemsSource automatically, triggering CollectionChanged.

```
<conversationalUi:RadChat ItemsSource="{Binding
ConversationItems}"  Author="{Binding Me}"/>
```

Typing Indicators

You can show participants who are currently typing using ObservableCollection<Author> in the ViewModel. Anyone in this collection will be shown by the RadChat control as current typing.

```
<conversationalUi:RadChat ItemsSource="{Binding ConversationItems}"Author="{B
inding Me}" >

    <conversationalUi:RadChat.TypingIndicator>

        <conversationalUi:TypingIndicator ItemsSource="{Binding
TypingAuthors}"  />

    </conversationalUi:RadChat.TypingIndicator>

</conversationalUi:RadChat>
```

Visit the Typing Indicators tutorial page for more information. You can also check out this Knowledge Base article, which explains how to have a real-time chat room experience with the indicators.

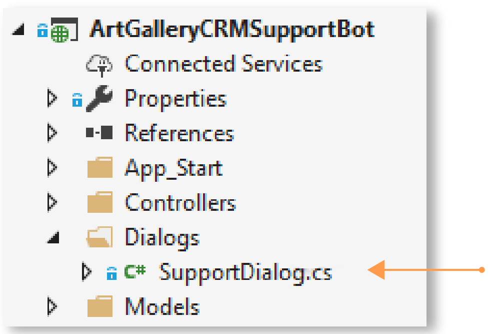
Message Styling

RadChat gives you easy, out-of-the-box functionality and comes with ready styles to use as-is. You can also customize the items to meet your branding guidelines or create great-looking effects like grouped consecutive messages. Visit the ItemTemplateSelector tutorial for more information.

Connecting the Bot to LUIS

You can run the application and confirm that the bot is echoing your messages. Then, connect the bot server logic to LUIS to ensure meaningful messages are returned to the user.

To do this, go back to the bot project and open the EchoDialog class. You could rename the class to better match its function. For example, the CRM demo dialog is named "SupportDialog".



In the MessageReceived Task, you can see what the Xamarin.Forms user typed into the RadChat control. The next step is to send the input to LUIS for analysis and determine how to respond to the user.

Microsoft has made it easy to use Azure Services by providing .NET SDK for LUIS. Install the **Microsoft.Azure.CognitiveServices.Language.LUIS.Runtime** NuGet package to the bot project:

Microsoft.Azure.CognitiveServices.Language.LUIS.Runtime by Microsoft Provides API functionalities for consuming the Microsoft Azure Cognitive Services LUIS Runtime API.

Once installed, you can go back to the dialog class and replace the echo logic with a LUISRuntimeClient that sends the user's message to LUIS. The detected Intents will be returned to your bot and you can then choose how to respond.

Here's an example that takes only the highest scoring Intent and thanks the user:

```
public async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<IMessageActivity> argument)

{    var userMessage = await argument;
     using (var luisClient = new LUISRuntimeClient(newApiKeyServiceClientCre
dentials("YOUR LUIS PRIMARY KEY")))
```

```
    }

        {

            luisClient.Endpoint = "YOUR LUIS ENDPOINT";

            // Create prediction client

            var prediction = new Prediction(luisClient);

            // Get prediction from LUIS

            var luisResult = await prediction.ResolveAsync(

                appId: "YOUR LUIS APP ID",

                query: userMessage,

                timezoneOffset: null,

                verbose: true,

                staging: false,

                spellCheck: false,

                bingSpellCheckSubscriptionKey: null,

                log: false,

                cancellationToken: CancellationToken.None);



            // You will get a full list of intents. For the purposes of this
    demo, we'll just use the highest scoring intent.

            var topScoringIntent = luisResult?.TopScoringIntent.Intent;



            // Respond to the user depending on the detected intent



            if(topScoringIntent == "Product")

            {

                // Have the Bot respond with the appropriate message

                await context.PostAsync("I'm happy you asked about products!");

            }

        }

    }
```

The **Primary Key**, **Endpoint** and **LUIS App ID** values can be found on www.luis.ai. (different than the Microsoft App ID for the bot project). Learn more here Quickstart: SDK Query Prediction Endpoint.

The code above uses a very simple check for a Product intent and replies with a statement. If there is no match, no reply will be triggered. The takeaway here is that, even though you control what gets sent back to the user, LUIS helps you make intelligent decisions and customize the experience to the user's needs.

Install Telerik UI for Xamarin Controls

Get Application Source Code

Download Telerik ERP demo application for iOS, Android and Windows devices.

Telerik ToDo: How to Build a Mobile Task Management Application with Xamarin.Forms

The Telerik ToDo sample application aims to mimic a real-world consumer-facing application. It uses a combination of various technologies and is developed in line with the latest trends in mobile development architecture.

Overview of the Technologies and Frameworks

- Entity Framework Core: A lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology. Entity Framework is the standard technology when it comes to .NET developers working with various database implementations

- SQLite: SQLite is the most used database engine in the world. It is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day. We chose SQLite to serve as the SQL database engine for the ToDo application

- MVVMFresh: A super-light MVVM framework created specifically for Xamarin.Forms. It is designed to be easy, simple and flexible. Some of the important features the framework provides are:
  - PageModel to PageModel navigation
  - Automatic BindingContext wiring
  - Built-in IOC containers
  - Automatic wiring of Page events and many more

- Telerik UI For Xamarin: Offers high-quality Xamarin.Forms UI components and Visual Studio item templates to enable every developer, regardless of their experience, to build professional-looking modern mobile applications for iOS, Android and UWP.

Application Structure

The different modules of this sample Xamarin.Forms application are separated in folders with names that are descriptive and quite standard for projects of such size. Some of the more important folders we have introduced are:

- Models: This is where the business objects live

- PageModels: The classes used as BindingContext for each separate Page (or view) are located here

- Pages: Hosts the different views/pages that the application will present

- DataAccess: The classes related to the database used to store the data of the application are in this folder

- Controls: Contains some small additional custom controls used within the pages

- Services: The application currently contains just a single ToDoService that takes care of storing and updating the data used within the application

A Lightweight MVVM Framework Implementation for Xamarin.Forms

The MVVM pattern is the standard for applications of any scale in the Xamarin world. The Telerik ToDo project relies on the MVVMFresh framework to keep the model and pages organized. The framework is famous in the Xamarin community for being easy and straightforward to use, which makes it a perfect fit for the purpose of our project.

In order to connect the views and their models, all you need to do is follow the convention in the naming—each Page should have a corresponding PageModel. For example, the BindingContext of the MainPage will automatically be set to be an instance of the MainPageModel class.

The navigation within the application is pretty simple as well. You can use the CoreMethods property of the FreshBasePageModel class, which is the base class for all the PageModels within the application, to navigate between models.

For more information about the different features the framework supports, you can refer to its documentation here—MVVMFresh.

Data Access

As mentioned earlier, we are using Entity Framework Core and its built-in ability to easily communicate with an SQLite database to power the data access capability of this Xamarin.Forms sample application. For the purpose, a couple of NuGet packages are installed:

- Microsoft.EntitiFrameworkCore
- Microsoft.Entity FrameworkCore.SQLite

The classes needed for EF are all located in the DataAccess folder. Note that there is a DTO folder where the Data Transfer Objects (the objects used to send data between our application and the underlying database) are located. They are almost identical to the objects within the Models folder and take care of the information we would like to keep in our database.

The DomainMapper class contains some methods to convert between these identical objects.

Additionally, we have added a few sample items in the database so that users can see exemplary to-do tasks when they open the application. The sample data creation can be viewed in the SeedData class.

The final class from the DataAccess layers is the TodoItemContext class, which extends the Entity Framework DbContext class. As advised in the official documentation, the DbContext class can be viewed as a session between your application and the database and can be used to query the database or update it with new information when necessary. It consists of DbSets for the different DTO objects we have. This DbContext is extensively used in the ToDoService class where the actual methods of querying and updating the database are implemented.

As the file systems on each mobile technology differ, we have introduced an IDbFileProvider interface that requires its implementations to set up a single method—GetLocalFilePath. A different implementation of the interface is provided by classes in each separate platform in order to create the database in the correct location. The Xamarin.Forms DependencyService approach is used to obtain the different locations according to the platform on which the application is running.

Creating the database can be found in the App.xaml.cs file: Ю

```
public static DataAccess.TodoItemContext CreateDatabase()

{

    IDbFileProvider dbFileProvider = DependencyService.
Get<IDbFileProvider>();

    if (dbFileProvider == null)

        return null;


    // Database

    string dblocation = dbFileProvider.GetLocalFilePath(“tododb.db”);

    System.Diagnostics.Debug.WriteLine($”Database location: {dblocation}”);

    DataAccess.TodoItemContext ctx = DataAccess.TodoItemContext.
Create(dblocation);

    return ctx;

}
```

Services

Back to the earlier technology overview, the Application Services consist of a single ToDoService that takes care of adding, deleting, and updating the to-do items within the application. It can also generate a list of the existing items.

Here are a couple of examples of methods that the service provides. The first is used to get a specific to-do item from the database and the second one to add a to-do item:

```
public TodoItem GetTodoItem(int id)
{
    return context.TodoItems
        .Where(c => c.ID == id)0
        .Include(c => c.Category)
        .Include(c => c.Alert)
        .ThenInclude(c => c.Alert)
        .Include(c => c.Recurrence)
        .ThenInclude(c => c.Recurrence)
        .Include(c => c.Priority)
        .SingleOrDefault()?.ToModel(true);
}
public async Task<TodoItem> AddTodoItemAsync(TodoItem newItem)
{
    var dto = newItem.ToDTO();
    var entity = this.context.TodoItems.Add(dto);
    try
    {
        await this.context.SaveChangesAsync();
    }
    catch (Exception e)
    {
        System.Diagnostics.Debug.WriteLine(e);
        throw e;
    }
    TodoItem storedItem = GetTodoItem(entity.Entity.ID);
    MessagingCenter.Send<ITodoService, TodoItem>(this, ActionAdd, storedItem);
    return storedItem;
}
```

Frontend Implementation: UI Layer and Telerik UI for Xamarin Controls

We have integrated multiple controls from the Telerik UI for Xamarin suite in the application to achieve a modern look, crafted by our design team.

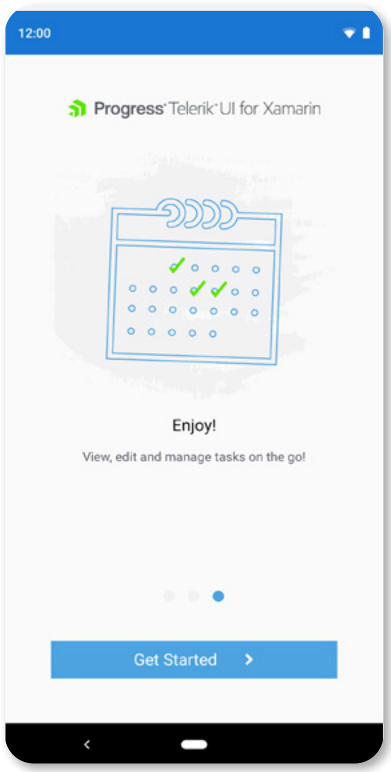
Embedding the controls is easy and straightforward. Simply follow the Telerik UI for Xamarin documentation. You can trust the documentation is always up to date and contains the right information for you.

Introduction Page and RadSlideView

The Welcome screen greets the user. The page utilizes the **RadSlideView control,** which directly sets an ItemTemplate. It is a great fit for any splash screen scenario or views where a sliding content is required. The code for bootstrapping the SlideView element is small and tight:

```
<telerikPrimitives:RadSlideView x:Name="slideView"
                        Grid.Row="1"
                        Margin="0,38,0,34"
                         ItemsSource="{Binding Slides}"
                        ShowButtons="False"
                        IndicatorColor="#F2F2F3"
                        SelectedIndicatorColor="#4DA3E0">

          <telerikPrimitives:RadSlideView.ItemTemplate>

              <DataTemplate>

                  <StackLayout Orientation="Vertical"Mar gin="0,0,0,50"
                   Spacing="0">

                      <Image Source="{Binding Image}"Aspect="AspectFit">

                          <Image.HorizontalOptions>

                              <OnPlatformx:TypeArguments="LayoutOptions"
                              Default="FillAndExpand">

                                  <On Platform="UWP"Value="CenterAndExpand"/>

                              </OnPlatform>

                          </Image.HorizontalOptions>

                          <Image.WidthRequest>

                              <OnPlatformx:TypeArguments="x:Double"
                              Default="-1">

                                  <On Platform="UWP"Value="300" />

                              </OnPlatform>
                      </Image.WidthRequest>

                      </Image>

                      <Label Text="{Binding Title}"HorizontalTextAlignment="
Center" TextColor="Black" FontSize="Medium"Margin="0,15,0,0" />

                      <Label Text="{Binding Description}"HorizontalText|Align
ment="Center" TextColor="#5C5C68"  Margin="0,14,0,0"/>

                  </StackLayout>

              </DataTemplate>

          </telerikPrimitives:RadSlideView.ItemTemplate>

      </telerikPrimitives:RadSlideView>
```

Setting the **ItemsSource**, the **ItemTemplate** and some additional properties of the control will make your mobile application look professional. This is how the Welcome page looks when the app is initially opened:


12:00
Progress Telerik UI for Xamarin
Enjoy!
View, edit and manage tasks on the go!
Get Started

Main Page and RadListView

From the Welcome page, users are guided to the Main page, where they can see the list of their do-to items. By default, a list in grid format will show the categories in which the to-do items are grouped. The view is achieved by solely including a RadListView control. As one of the more complex and feature-rich controls in the suite, it can be highly customized to achieve a distinguished look. Here are some of the features we used in this particular view:

- Grid Layout

```
<telerikDataControls:RadListView.LayoutDefinition>
    <telerikListView:ListViewGridLayout HorizontalItemSpacing="0"
                            ItemLength="100"
                            SpanCount="2"
                            VerticalItemSpacing="0" />
</telerikDataControls:RadListView.LayoutDefinition>
```

- Custom Item Styles

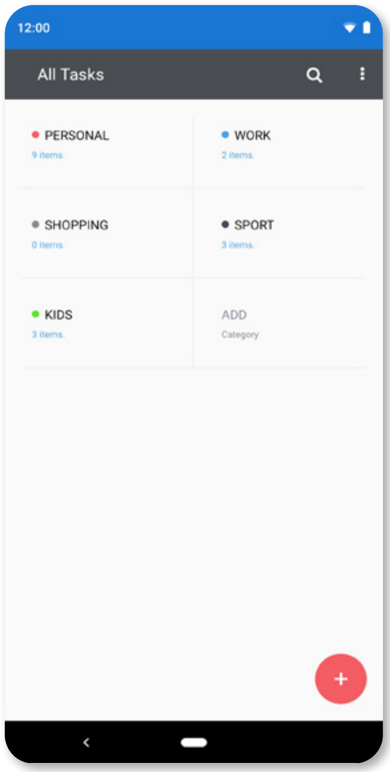
```
<telerikDataControls:RadListView Grid.Row="1"
                      ItemsSource="{Binding Categories}"
                      ItemStyle="{StaticResource UnifiedItemStyle}"
                      SelectedItemStyle="{StaticResource UnifiedItemStyle}"
                      PressedItemStyle="{StaticResource UnifiedItemStyle}"
                      SelectedItem="{Binding SelectedCategory, Mode=TwoWay}">
```

- ItemTemplateSelector

```
  <telerikDataControls:RadListView.ItemTemplateSelector>
      <templateSelectors:RadListViewItemTemplateSelector>
          <templateSelectors:RadListViewItemTemplateSelector.
CategoryTemplate>
              <DataTemplate>
                  <telerikListView:ListViewTemplateCell>
                      <telerikListView:ListViewTemplateCell.View>
                          <-- Template for the existing category items -->
                      </telerikListView:ListViewTemplateCell.View>
```

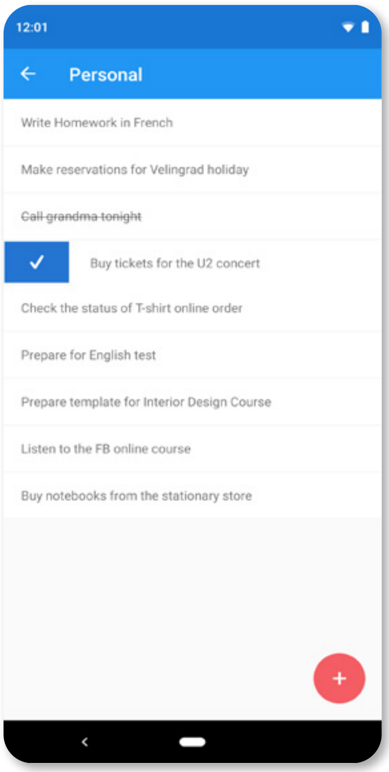
```
                    </telerikListView:ListViewTemplateCell>

                </DataTemplate>

            </templateSelectors:RadListViewItemTemplateSelector.
CategoryTemplate>

            <templateSelectors:RadListViewItemTemplateSelector.
NewCategoryTemplate>

                <DataTemplate>

                    <telerikListView:ListViewTemplateCell>

                        <telerikListView:ListViewTemplateCell.View>

                           <-- Template for the new category item -->

                        </telerikListView:ListViewTemplateCell.View>

                    </telerikListView:ListViewTemplateCell>

                </DataTemplate>

            </templateSelectors:RadListViewItemTemplateSelector.
NewCategoryTemplate>

        </templateSelectors:RadListViewItemTemplateSelector>

</telerikDataControls:RadListView.ItemTemplateSelector>
```

Here is how the different categories will be visualized when using such setup:



You can check the full setup of the control directly in the application source code available in the public GitHub repository.

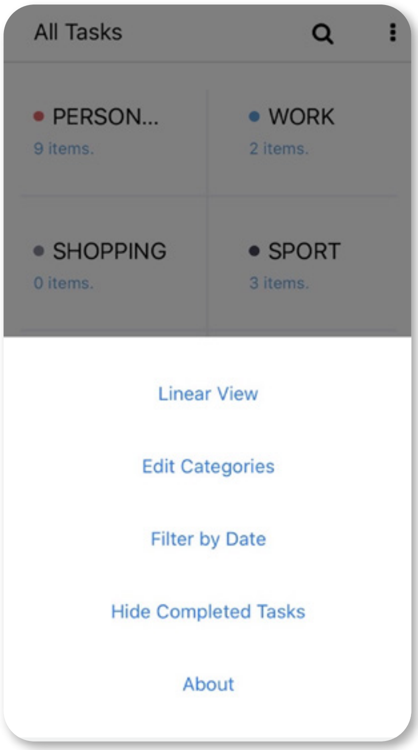
Clicking on each separate category will open a new screen listing the to-do items marked with this specific category. The new view is accomplished through a RadListView component, this time in a LinearLayout mode. On this page, users can mark an item as done, which will in turn cross off the item or delete it. The RadListView swiping feature facilitates these interactions:


12:01
Personal
Write Homework in French
Make reservations for Velingrad holiday
~~Call grandma tonight~~
Buy tickets for the U2 concert
Check the status of T-shirt online order
Prepare for English test
Prepare template for Interior Design Course
Listen to the FB online course
Buy notebooks from the stationary store

Furthermore, you can navigate to the different to-do items and edit their specific properties from this screen or by simply clicking on the item.

Alternative Look with RadTreeView

The application contains all the necessary actions to allow users to add and edit different items and categories. However, we decided to provide an additional option for clients to customize the list view. Instead of using the default grid-like view, the app provides an option to show the items in a list with categories as groups. In order to show this view, simply click on the hamburger menu on the right side where some additional options will appear:



The menu is set as the DrawerContent of a RadSideDrawer control. This approach is used to easily show the different options with a button click or through a swiping gesture.

We added a RadTreeView component to order the categories and items hierarchically. Clicking the different categories will collapse/expand the list below. When setting up the TreeView component, it is important to apply the correct TreeViewDescriptors:

```
<telerikDataControls:RadTreeView x:Name="treeView" Grid.Row="1"
                                 ItemsSource="{Binding Categories}">
    <telerikDataControls:RadTreeView.Descriptors>
        <telerikDataControls:TreeViewDescriptor TargetType="{x:Type
models:Category}" DisplayMemberPath="Name" ItemsSourcePath="Items">
            <telerikDataControls:TreeViewDescriptor.ItemTemplate>
                <templateSelectors:RadTreeViewItemTemplateSelector>
                    <templateSelectors:RadTreeViewItemTemplateSelector.
CategoryTemplate>
                        <DataTemplate>
                         <!--Category Item Template-->
                        </DataTemplate>
                    </templateSelectors:RadTreeViewItemTemplateSelector.
```

```
CategoryTemplate>
                    <templateSelectors:RadTreeViewItemTemplateSelector.
NewCategoryTemplate>
                        <DataTemplate>
                            <!--New Category Item Template-->
                        </DataTemplate>
                    </templateSelectors:RadTreeViewItemTemplateSelector.
NewCategoryTemplate>
                </templateSelectors:RadTreeViewItemTemplateSelector>
            </telerikDataControls:TreeViewDescriptor.ItemTemplate>
        </telerikDataControls:TreeViewDescriptor>
        <telerikDataControls:TreeViewDescript
or TargetType="{x:Type models:TodoItem}" DisplayMemberPath="Name">
            <telerikDataControls:TreeViewDescriptor.ItemTemplate>
                <DataTemplate>
                 <!--ToDo Item template-->
                </DataTemplate>
            </telerikDataControls:TreeViewDescriptor.ItemTemplate>
        </telerikDataControls:TreeViewDescriptor>
    </telerikDataControls:RadTreeView.Descriptors>
</telerikDataControls:RadTreeView>
```

We defined separate descriptors for the Category and ToDoItem business objects. We also set a custom ItemTemplateSelector for the Category as we aim for an appearance similar to the RadListView.

If you explore the demo application, you will notice that we used the RadListView component on several occasions throughout the application as the control is versatile and comes in hand in many common mobile scenarios. Other elements from the Telerik UI for Xamarin suite used in the application are the RadButton, RadBorder and RadPopup controls.

Install Telerik UI for Xamarin Controls

Get Application Source Code

Download Telerik ERP demo application for iOS, Android and Windows devices.

Conclusion

Building cross-platform mobile applications with Xamarin has its challenges and the best way to overcome them is by following the best available practices and learning by example. The purpose of this whitepaper was to introduce you, the Xamarin or .NET developer, to several real-world Xamarin.Forms line-of-business applications and how they were built from the ground up. We hope you found this resource useful for your future Xamarin development endeavors.

If you have any questions or feedback regarding the demo applications or the Telerik UI for Xamarin controls suite, feel free to reach out to our team. We'll be glad to hear from you!