ISSUE 13 | JULY - AUG 2014

DNCMagazine

www.dotnetcurry.com

powerful file format APIs

Working with Files?

Convert Print Create Combine Modify

Aspose.Words
DOC, DOCX, RTF, HTML, PDF, XPS & other document formats.

Aspose.Pdf
PDF, XML, XLS-FO, HTML, BMP, JPG, PNG & other image formats.

Aspose.Cells
XLS, XLSX, XLSM, XLTX, CSV, SpreadsheetML & image formats.

Aspose.Slides
PPT, PPTX, POT, POTX, XPS, HTML, PNG, PDF & other formats.

Aspose.Email
MSG, EML, PST, EMLX & other formats.

Aspose.BarCode
JPG, PNG, BMP, GIF, TIFF, WMF, ICON & other image formats.

Aspose.Imaging
PDF, BMP, JPG, GIF, TIFF, PNG, PSD & other image formats.

Aspose.Tasks
XML, MPP, SVG, PDF, TIFF, PNG, CSV, MPT & other formats.

Aspose.Diagram
VSD, VSDX, VSS, VST, VSX & other formats.

Aspose.Note
ONE, PNG, JPG, BMP, GIF & PDF

... and more!

100% Standalone - No Office Automation

.NET Libraries | Java Libraries | Cloud APIs | Android Libraries

Download Now

ASPOSE
Your File Format APIs

US Sales: +1 888 277 6734
sales@aspose.com

EU Sales: +44 141 416 1112
sales.europe@aspose.com

AU Sales: +61 2 8003 5926
sales.asiapacific@aspose.com

INFRAGISTICS
DESIGN / DEVELOP / EXPERIENCE

MELISSA DATA®

amazon.com gift card $100.00

NEVRON

amazon.com gift card $100.00

To participate in the Giveaway

CLICK HERE

CONTENTS

EDITOR'S NOTE

> Without **YOU**, this magazine would not exist! On behalf of the entire DotNetCurry (DNC) Magazine team and authors, **Thank You** for reading us.

It's overwhelming to think how far this magazine has come in the past two years, and how rapidly Microsoft and its technologies have changed in this short span. Microsoft realizes that its products and the framework have to evolve in order to stay relevant in a world, where apps have to target multiple platforms. Their strategy is very clear - a mobile-first, cloud-first world!

Satya Nadella, the CEO of Microsoft recently said that "Microsoft can offer developers the tools they need to compete in an increasingly cross-platform world". This statement was echoed in the recently made announcements during premier technology events like Microsoft Build and TechEd. The 'Write Once, Run Everywhere' mantra, Universal App development strategy and Microsoft's demonstrations of their vNext tools and technologies, illustrates that the company is moving in the right direction.

As Microsoft transitions many of its most popular products to the cloud, we as developers have to be prepared to learn and stay ahead of the curve! This year we @DotNetCurry plan to stay in sync with Microsoft's strategy and will offer you different perspectives, detailed walkthroughs, as well as interesting peeks into these exciting technologies, as it metamorphizes.

Amidst this rush, there's always time to pause, thank and celebrate this journey with you. We begin by thanking our fabulous authors who have contributed some excellent .NET content past one year. To thank YOU for being a reader and for your countless accolades and constructive criticisms, we are doing a Giveaway of .NET Software, Kindle Books and Gift cards. Make sure you participate to win these cool goodies. Come, celebrate with us!

Suprotim Agarwal
Editor in Chief

CREDITS

Editor In Chief Suprotim Agarwal
suprotimagarwal@dotnetcurry.com

Art Director Minal Agarwal
minalagarwal@a2zknowledgevisuals.com

Contributing Writers Craig Berntson, Filip Ekberg, Mahesh Sabnis, Pravinkumar Dabade, Ravi Kiran, Subodh Sohoni, Suprotim Agarwal, Vikram Pendse

Reviewers: Alfredo Bardem, Gil Fink, Mahesh Sabnis, Nishanth Anil, Suprotim Agarwal

Next Edition 1st Sep 2014
www.dncmagazine.com

INTRODUCING XAMARIN

BUILDING A REST API BASED HACKERNEWS APPLICATION THAT WORKS ON IOS, ANDROID AND WINDOWS PHONE

WHAT IS XAMARIN AND WHY SHOULD I CARE?

The past 5-7 years has been extremely interesting in the mobile world, especially considering the release of iPhone, Android and Windows Phone. There has been a lot going on in the mobile spectrum even outside these three different brands. These three in particular share something though; they're all easy to create applications for. This is of course subjective, it's easier for some than for others, but if we compare the way we create native mobile applications for these three different platforms, to what we saw 10 years ago; this is easier.

Over the years we've seen a lot of interesting abstractions that have made the development even easier. Consider that you have an application that needs to target all the major platforms; iOS, Android and Windows Phone. In general that meant to either develop one application natively for each platform or develop the application using a shared code principle or a wrapper such as PhoneGap. Both of these strategies for cross-platform development have their ups and downs. The problem with the first approach of creating one application natively per platform is that we need a team that specializes in each platform. That means three different teams understanding the same acceptance criteria, hence three times as expensive, in general that is.

The benefit of going all-in on Native, is that we get a specialized user interface and it performs fast. Compared to the approach where we use PhoneGap or similar, to write an application in HTML, CSS and JavaScript that runs everywhere; the native approach wins on performance and feel. You can make an application good enough using PhoneGap, Sencha or similar, but it will never beat the native experience. However, write once – run everywhere has its benefits. You have *one* code base. This means one team developing the application, one team understanding the acceptance criteria and this means not as expensive.

XAMARIN TO THE RESCUE

Up until recently, going native has meant to develop three different applications, using three different languages and that has made it really difficult for companies to select more than 1-2 platforms to target. With Xamarin, this is no longer necessary! What Xamarin does is that it introduces a third strategy for cross-platform development; ***write in one language, run everywhere and customize the user interface***.

The way it works is that Xamarin will let you write the code in C#, a language we all love, then target all the major platforms - iOS, Android, Windows, Mac, Windows Phone. Most of these platforms don't run the executables C# normally compiles to, so what Xamarin does is that it compiles it to the native code that each platform understands and produces the necessary binary. This means for iOS we have the *IPA* files and for Android we have *APK*.

This means that the code written with this approach, will run natively on each platform and perform just as well as if it was written in the language that the platform normally prefers; i.e. Objective-C for iOS and Java for Android.

In order to get a *native* user interface though, and with native user interface I refer to the fact that an iOS user doesn't want an Android experience, we have a different user interface layer per platform. This means that the iOS, Android and Windows Phone project will be different. This is also what defines what type of executable to produce when the project itself is compiled.

Up until recently there's been a goal of being able to share about 90% of the code between the different platforms. This is done by taking advantage of something called *Portable Class Library (PCL)*. However, Xamarin has upped their game and released something called `Xamarin.Forms`, which enables you to share up to 100% of the code.

One language, one team, native performance, all major platforms – what else could we ask for when creating mobile applications?

CODE SHARING PRINCIPLES

Being able to share up to 100%, or even 90% among three different platforms is a *great* achievement. It will also mean there is less code to maintain, given that you didn't write more code to be able to share of course. Consider all the business logic that you have in your applications written earlier. You may have ways to process users such as adding or removing. You might have ways to process invoices, reports, privileges and what not. Are any of those specific to iOS, Android or Windows Phone? Hopefully not, thus having this shared in a common class library (PCL) means *we don't have to write it once per platform.*

Other things are platform specific and should be, such as reading and writing to the storage. This works differently on iOS, Android and Windows Phone, so how do we write sharable code for this? If we have a lot of logic around storing and processing stored data, we don't want to keep all that sharable logic, redundantly in each platform specific project. The answer is *Interfaces*!

You're developing an application that handles lots of data, you want to cache the data and only download new data after 1 – 2 days, or whatever expiry period the data might have. You want to hide the caching mechanism in your repository, but the loading and saving to the device storage is platform specific, so what we need is an interface that we can implement in each of the platform specific projects.

Let's say that we have the interface `IStorageHandler`, each of our projects can now have an implementation of this interface: `iOS.StorageHandler`, `Droid.StorageHandler` and `WindowsPhone.StorageHandler`. When the application starts, we simply *inject* the correct storage handler into the repository and it will use the platform specific code. Because after all, the interface is just a contract telling us what something can do, it's up to us to define the implementation where it matters.

Sometimes though, there's no common interface to use, which

might be problematic. You might want to trigger Azure Mobile Service authentication from a shared PCL, but there is no PCL version of Azure Mobile Service; however it does exist for all the platforms. What you can do in this case is to inject an `Action` or a `Func` to trigger the authentication. I prefer the interfaces, but that's just me.

CREATING A USER INTERFACE FOR EACH PLATFORM

While sharing the logic for the business layer might seem obvious, the question about what we do about the user interface arises. It's not immediately evident that we have to separate each implementation of the user interfaces for each platform. However this is true with some modification, as stated previously, the goal is to share up to 100% of the code. Previously this is something we didn't do. As each platform have their own specific user interface design principles, it's been more convenient not to try and abstract this layer. Not just because it's more convenient to make the native user interface using the approach that you'd rather do natively - Android XML for Android, Interface Builder/Storyboards for iOS and XAML for Windows Phone. Also because following this pattern separated the development where it mattered. Without the separation, it's easy to get stuck in the *write once, run everywhere* paradigm where we have to conform to a lowest common denominator.

With the release of Xamarin 3, you're not obligated to go with this approach. You can create a shared user interface layer by defining the user interface in code - C# or XAML. It's not necessary to have the entire application written with `Xamarin.Forms`, if it happens that there's something you can't share, user interface wise, you can create that specifically for each platform using the approach that one has been able to do before. Consider for just a second how powerful it is to be able to create a mobile application using *one* language, getting a native experience both performance and user interface wise, without letting the users down. It's perfect!

SHARING A USER INTERFACE WITH XAMARIN.FORMS

Fortunately for us with the latest version, Xamarin 3, we get `Xamarin.Forms` installed with some good project templates. Developing applications for iOS, Android and Windows Phone, in my opinion, is easiest on a Mac. You might wonder why is that so? Simple: we can't compile iOS applications on Windows due to license restrictions. However, Xamarin solves this for us by letting us connect to a Mac, be it the same machine using parallels or a machine on the network. We can use the ***Xamarin Build Host***. This will let you connect Visual Studio to the build host, in order to enable build and debug from Windows. Personally I prefer doing this in parallels but it could just as well be a Mac machine on your network, or why not *MacInCloud*!

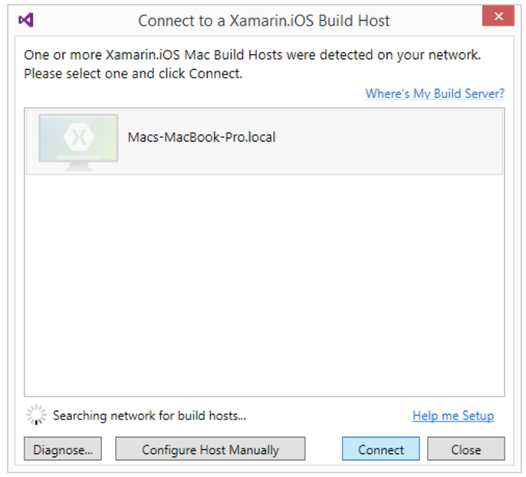


Figure 1: Xamarin Build Host

In Visual Studio, after of course having installed Xamarin 3, you will find the new project templates in the Mobile Apps selection. You can either select to use portable class libraries, or a shared project. The shared project is a new paradigm that was introduced first with Visual Studio 2013 Update 2 for Universal Apps.

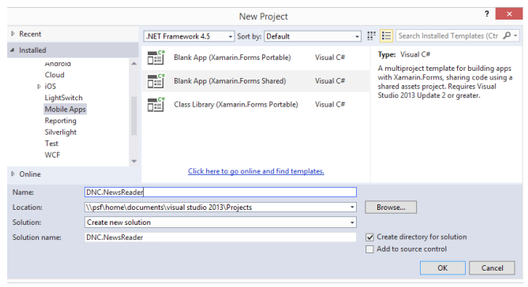


Figure 2: Xamarin Project Template in VS 2013 Update 2

Selecting the shared option will create 4 projects:

- A Shared project containing the shared UI
- Android project setting up the activity to use the shared UI
- iOS project setting up the view controller to use the shared UI
- Windows Phone project setting up the main page to use the shared UI

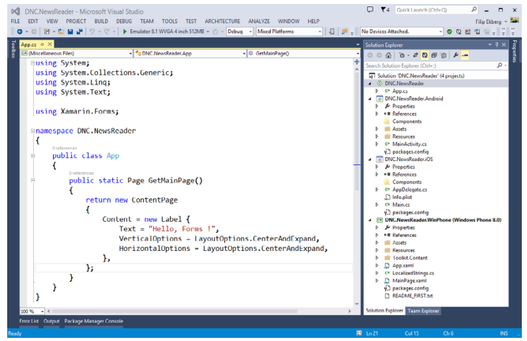


Figure 3: Shared Code

The shared code in this template application simply sets up a new page; you might recognize the user interface principles from Windows Phone! Here we're simply setting up a new content page with a label in it.

Figure 4: Application running in an older iOS simulator

If we instead mark the iOS project as the start-up project, and select the platform to be the iOS Simulator, you may now run the application and see that it successfully deploys to iOS!

We of course want to change this user interface, but this gives you a basic idea that we have the code for the interface shared! All the different platforms create this page differently. The shared code is accessible via `App.GetMainPage()` and the different platforms uses it differently to set it up.

IOS

In the App Delegate for iOS, where we setup the main view controller, we simply create and use the shared user interface like this:

```
Forms.Init();
App.GetMainPage().CreateViewController()
```

ANDROID

In the main activity for android, we set it up differently compared to iOS, as both the platforms are fundamentally different:

```
Xamarin.Forms.Forms.Init(this, bundle);
SetPage(App.GetMainPage());
```

WINDOWS PHONE

For Windows Phone what happens in the main page is that it converts the page returned from the shared user interface code, to a real `UIElement`. It then sets the current page content to whatever that element represents:

```
Forms.Init();
Content = DNC.NewsReader.App.GetMainPage().
ConvertPageToUIElement(this);
```

Note the call to `Forms.Init(). Forms.Init()` must be called before making any calls to Forms API. This step enables the device (iOS, Android, Windows Phone) specific implementation of Xamarin.Forms to be available globally in the application.

CAPABILITIES AND POSSIBILITIES

Frankly I think you can solve about anything using `Xamarin.Forms` without too much diversity. There are so many different layouts and controls available out of the box such as: `Stack-`, `Grid-`, `Absolute-`, `RelativeLayout`, `Button`, `Image`, `Map`, `Editor`, `WebView`, `ActivityIndicator` and much more. All these components and layouts make up for most of the things that you could possibly want to use; unless you are creating a game of course, but then in that case, you might be looking at other alternatives such as *Unity*.

Of course what we have seen so far is just a very simple example, but with some imagination, you could see the endless possibilities and the hours, money and headaches saved by going this route.

CONSUMING REST APIS

Working with data is central to almost any application and a lot of our data today is accessible over HTTP. A REST API is just a fancy way of saying that we expose an API over HTTP, meaning that we can perform `GET`, `POST`, `PUT` and `DELETE`. Let's not worry too much about the HTTP verbs but only focus on using `GET` for this example. Here's what I'd like to achieve, see it as the acceptance criteria and allow it to be a bit vague:

The application should work on iOS, Android and Windows Phone with a minimalistic UI. The application uses an API to fetch news articles from HackerNews; which is publicly accessible by a third party. When listing news, show the title. When clicking the article, show the article content.

As you may have noticed, the *shared project* does not contain any references; this is because it relies on *File Linking*. This means that if we want to use certain libraries in our shared project, we need to install the reference in each UI layer.

However in our case, let's create a portable class library where we can interact with our API, that way we don't have to install the NuGet package in all the projects! We just need to reference our PCL to them.

I've created a new portable class library that I call *Shared*:

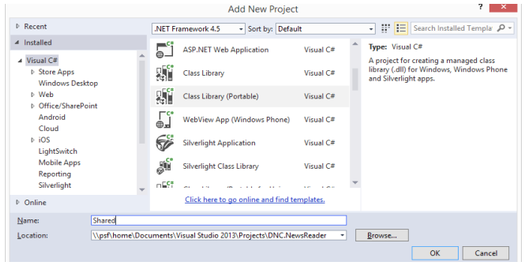


Figure 5: Portable Class Library

We're going to start off by installing Microsoft.Net.Http to get access to the HttpClient, which we will use in order to fetch the news from the HackerNews API. This is of course installed into our newly created PCL.

```
PM> Install-Package Microsoft.Net.Http
```

To make this work, you will also need to install JSON.NET.

```
PM> Install-Package Newtonsoft.Json
```

Within the shared library, the portable class library that I want to have is a class that I call *HackerNewsRepository*. This is my entry point from where I get the news. As a consumer of the repository, I don't really care where it comes from, if it's cached or what is going on – I just want the data!

```
public class HackerNewsRepository
{
    public async Task<IEnumerable<Entry>>
    TopEntriesAsync()
    {
        var client = new HttpClient();
        var result = await client.
        GetStringAsync("http://api.ihackernews.com/
        page/");
        return JsonConvert.
        DeserializeObject<Page>(result).Items;
    }
}
```

The repository is fairly straightforward, it relies on two models, performs a web request to a certain URL and returns the deserialized result from that. We are never exposing what we know as a *Page* inside the repository, but just an enumerable of *Entries*.

```
public class Entry
{
    public int Id { get; set; }
    public int Points { get; set; }
    public string PostedAgo { get; set; }
    public string PostedBy { get; set; }
    public string Title { get; set; }
    public string Url { get; set; }
}

public class Page
{
    public int? Next { get; set; }
    public IEnumerable<Entry> Items { get; set; }
}
```

From the example you saw above in Figure 4, where the application runs on iOS, I'm using an older simulator. This is just to show that this is backwards compatible and works good with older devices too! Going forward we will look at it using the latest iOS simulator.

Now that we have the possibility to fetch news from the *HackerNews* API, we want to make the necessary changes to the shared user interface code. Before we can do that, we need to have a reference to the shared portable class library in all the user interface layers.

Once that is done, we want to modify the *App* class to look like the following:

```
public class App {
    public static INavigation Navigation { get; private
    set; }
    public static Xamarin.Forms.Page GetMainPage()
    {
        var navigationPage = new NavigationPage(new
        HackerNewsPage());
        Navigation = navigationPage.Navigation;
        return navigationPage;
    }
}
```

There are a couple of distinct things to notice here. First, I had to change the method signature to use the full namespace for Page as this type now also exists in our shared portable class library. Secondly we are no longer returning just a content page, we are returning a navigation page and the navigation property from it is exposed as a property on the App. The `HackerNewsPage` is simply a class that inherits from `ContentPage`. This allows us to override certain methods, such as when the page appears.

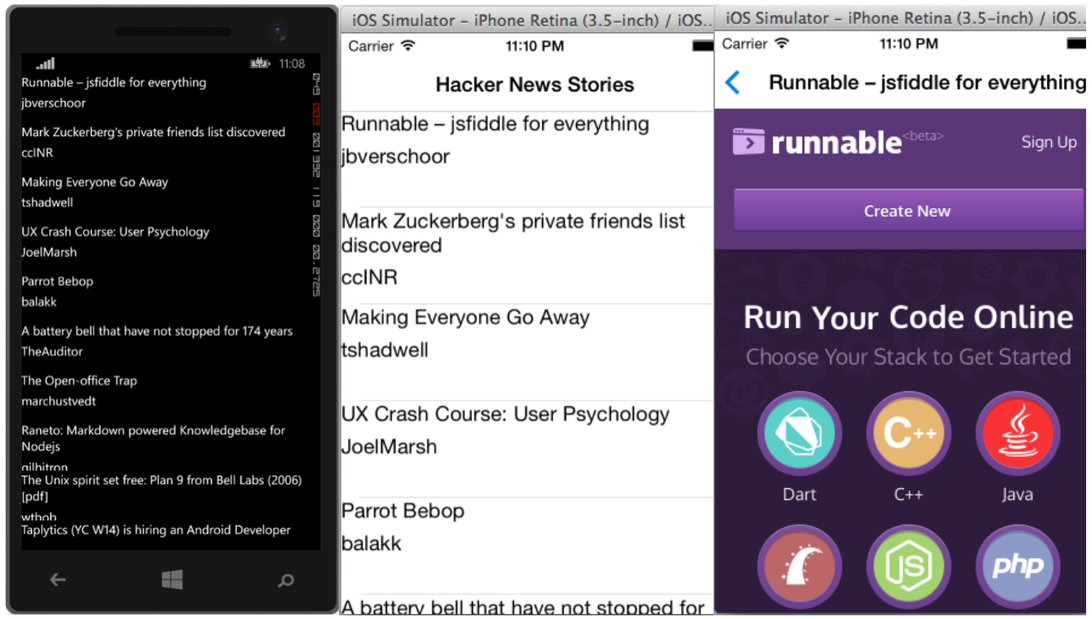
```
public class HackerNewsPage : ContentPage
{
    private ListView listView;
    public HackerNewsPage()
    {
        Title = "Hacker News Stories";

        listView = new ListView
        {
            RowHeight = 80
        };
        Content = new StackLayout
        {
            VerticalOptions = LayoutOptions.
            FillAndExpand,
            Children = { listView }
        };
    }
    protected override async void OnAppearing()
    {
        base.OnAppearing();
        var entries = await new HackerNewsRepository().
        TopEntriesAsync();

        listView.ItemTemplate = new
        DataTemplate(typeof(HackerNewsEntryCell));

        listView.ItemsSource = entries;
    }
}
```

Having an asynchronous void method *is generally bad*, but let's say that it's OK in this case and move on from that part. The code above creates the list view that we see; it starts loading the data when the view appears and populates the list view once the data has arrived. It uses a cell template that we need to define; in this cell template we can make use of data binding. This is also where we define what happens when we tap on one of the list view rows:

```
public class HackerNewsEntryCell : ViewCell
{
    public HackerNewsEntryCell()
    {
        var title = new Label();
        title.SetBinding(Label.TextProperty, "Title");

        var postedBy = new Label();
        postedBy.SetBinding(Label.TextProperty,
        "PostedBy");

        View = new StackLayout
        {
            Children = { title, postedBy }
        };
    }
```

```
    protected override void OnTapped()
    {
        base.OnTapped();
        var entry = BindingContext as Shared.Entry;
        var article = new WebView
        {
            Source = new UrlWebViewSource
            {
                Url = entry.Url,
            },
            VerticalOptions = LayoutOptions.
            FillAndExpand
        };

        App.Navigation.PushAsync(new ContentPage {
        Title = entry.Title, Content = article });
    }
}
```

Interestingly enough we are using the navigation that we previously made accessible in the application definition. We use this to push a new web view in a new content view, onto the stack of views. We can access the bound element that we clicked via BindingContext, hence this is how we can get the entry title and url. Running this gives us the native look and feel on different devices, as seen here:



CREATE THE UI USING XAML

If you rather prefer creating the user interface using XAML, that's equally easy! All you need to do is create a new Xamarin.Forms XAML file and add the corresponding code to that. Having seen the XAML from WPF, Silverlight, Windows 8 or Windows Phone helps to get you accustom to the syntax and the elements. All the controls that we have used translates directly into XAML and with just a slight modification on how we handle the tapping of an element, we can get this transformed into XAML. The following is all the code that goes into the HackerNewsPage.xaml in order to make it corresponding to what we had earlier. This replaces the HackerNewsPage and HackerNewsEntryCell that we saw previously:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/
forms" xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
x:Class="DNC.NewsReader.HackerNewsPage">
    <StackLayout>
      <ListView ItemsSource="{Binding}"
       ItemTapped="OnTapped"  RowHeight="80">
        <ListView.ItemTemplate>
          <DataTemplate>
            <ViewCell>
              <StackLayout>
                <Label Text="{Binding Title}"></Label>
                <Label Text="{Binding PostedBy}">
                </Label>
              </StackLayout>
            </ViewCell>
          </DataTemplate>
        </ListView.ItemTemplate>
      </ListView>
    </StackLayout>
</ContentPage>
```

As you can observe, we are taking advantage of bindings by binding to elements that we know are inside our collection of data. Now you're probably wondering what does the code behind looks like and honestly it's basically a copy and paste from what we had in the previous `OnTapped` method. The only difference is where we get our tapped element:

```
public void OnTapped(object sender, ItemTappedEventArgs
args)
{
    var entry = args.Item as Shared.Entry;
    var article = new WebView
    {
        Source = new UrlWebViewSource
        {
            Url = entry.Url,
        },
        VerticalOptions = LayoutOptions.FillAndExpand
    };

    App.Navigation.PushAsync(new ContentPage { Title =
    entry.Title, Content = article });
}
```

There is one more thing that we do have to change before we can get this to play well with our application. We have to tell the app startup to hook up the context that we want to bind.

We're going to do this by using an asynchronous event, in reality you want to unsubscribe to your events as well, but to keep the example easy, I'll make use of an anonymous, asynchronous method. We are also going to keep track of if the data is already loaded; this makes it much easier on the application. We no longer have to load the data each time the view appears:

```
public static Xamarin.Forms.Page GetMainPage()
{
    var page = new HackerNewsPage();
    var contentLoaded = false;

    page.Appearing += async (sender, args) => {
        if (!contentLoaded)
        {
            page.BindingContext = await new
            HackerNewsRepository().TopEntriesAsync();
            contentLoaded = true;
        }
    };
    var navigationPage = new NavigationPage(page);
    Navigation = navigationPage.Navigation;

    return navigationPage;
}
```

That is all there is to it, in order to convert the code we had previously used and instead use XAML! As you will see in Figure 7, running this on Android will look similar to what we've seen on the other platforms, but it has the Android feel. There's a bunch of things you can do to make the user interface look better of course, but that's outside the scope of this article.

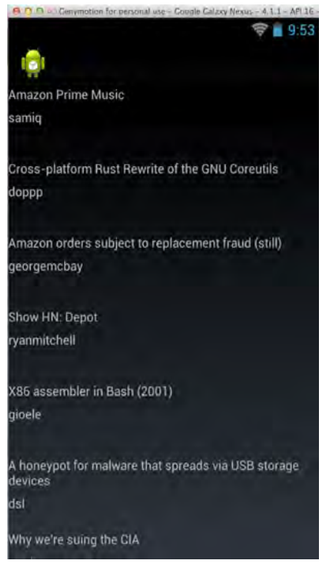


Figure 7: Android look and feel

CONCLUSION

Developing mobile applications seem to get easier by the day, especially with tools like Xamarin. There's no longer a valid reason to neglect one of the major platforms when developing a new application for yourself or for your business. Sharing all the user interface code in one library, sharing all business logic, one language, one team, and one set of experts; how else could it have possibly become better?

If you have a lot of time invested in a native platform, it might make sense to continue down that path, but keep in mind that Xamarin can help your transition by letting you create wrappers for native code. This means that you can develop using Xamarin for new applications, while still maintaining your native code-base as long as you need to. What are you waiting for?

Get started with your application for iOS, Android and Windows Phone today! ■

Download the entire source code from our GitHub Repository at bit.ly/dncm13-xamarin

Filip is a Microsoft Visual C# MVP, Book author, Pluralsight author and Senior Consultant at Readify. He is the author of the book C# Smorgasbord, which covers a vast variety of different technologies, patterns & practices.

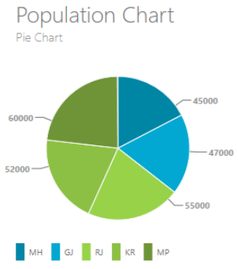
You can get the first chapter for free at: bit.ly/UPwDCd

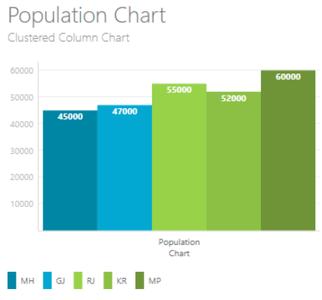
Follow Filip on twitter @fekberg and read his articles on fekberg.com

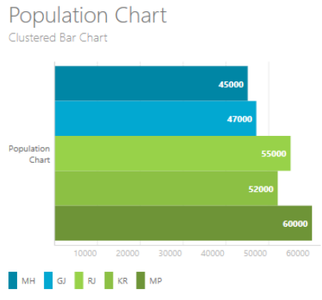
CREATING COOL WPF CHARTS

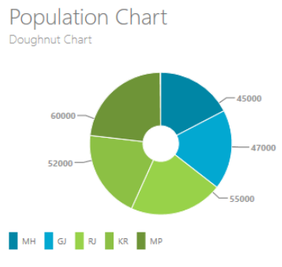
using the ModernUI Chart Library and WebAPI

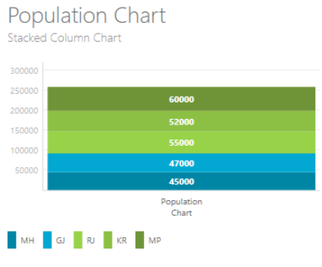
A picture is worth a thousand words! When working with business applications, a major challenge is to present data in a graphical form that is not only accurate, but also easy to interpret. Charts are a popular medium of presenting data and results graphically and it becomes the responsibility of the team to make use of the most suitable technology and chart type, which can be data bound to a variety of data sources.

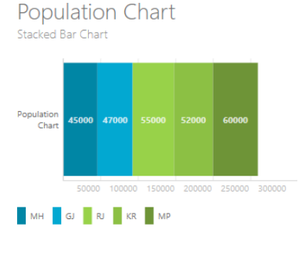












Windows Presentation Foundation (WPF), is a widely used popular .NET technology which can be used to create Rich UX experiences for the end-user and provides a variety of graphical representations based on the WPF graphical capabilities. One of its useful graphical representation is the 2-D Shape API containing Line, Polyline, Rectangle, Ellipse etc., using which graphical charts can be created. However creating a feature rich graphical representation can be a very tedious task. Implementing different chart-types like the Pie, Bar, Column etc. can be very time consuming. It would help if a library exists that can be used to add simple graphs and charts in your application. What would be even nicer is if the same library can be used for WPF as well as Windows Store apps too.

The ModernUI Charts is one such free library created by Torsten Mandelkow which can be used in your WPF applications and also conceptually supports Windows Store Apps as well as Silverlight applications.

Disclaimer: *The ModernUI Charts can be freely used in your application and is published under the Microsoft Public License which grants you these rights. However these charts are in Beta and has not been updated since May 2013. So it is advisable that before using it any production ready application, make sure you have tested in thoroughly. Having said that, our purpose of showcasing this free library is to give you a head start if you are planning to build your own chart controls. You can even build on top of the ModernUI Chart library and create something that matches your business requirements. If you want a professional charting control that is high performing and production ready, try LightningCharts from Arction (promotion link)*

The chart-types provided out-of-the-box in the ModernUI Charts library are as listed here:

- PieChart
- BarChart
  - ClusteredBarChart
  - StackedBarChart
  - StackedBarChart100Percent
- ColumnChart
  - ClusteredColumnChart
  - StackedColumnChart
  - StackedColumnChart100Percent
- Doughnut Chart
- Radial Gauge Chart

That is a quite a nice list for something that is available free of cost!

CREATING AN APPLICATION USING THE MODERNUI CHART LIBRARY

INSTALLING THE MODERNUI CHART

Follow these simple steps to install the ModernUI chart library in your application

Step 1: Download the solution project for the Modern UI Chart from the following link http://modernuicharts.codeplex.com/

Step 2: Open the solution in Visual Studio 2012/2013 and build it. This should give you the ModernUI Chart libraries for WPF, Silverlight and Windows 8 Application development. For our demo, we will only be using the ModernUI Chart library for WPF and for that, we will be referencing it in a WPF 4.5 project which we will create shortly.

Note: *The ModernUI Chart is still in its Beta, so for all custom requirements you need to extend the library on your own. The source code for it is available and for those who are really interested, that's all that is needed to get started!*

Obtaining a Data Source - AdventureWorks2012 Database

For our application, we will be using an Adventure Works database to give you the feel of a real application. The Database can be downloaded from the following link: http://msftdbprodsamples.codeplex.com/releases/view/55330

Once you have downloaded the AdventureWorks2012_Data.mdf file, integrate it with a SQL Server Instance on your server. We are using SQL Server 2012. We will be specifically using the *SalesTerritory* table from the database for displaying Sales data using ModernUI Charts.

CREATING AN APPLICATION USING WEB API AND WPF

In our application, we will be using ASP.NET WEB API to fetch our sales data and make it available to our WPF Application for displaying charts. Follow these steps:

Step 1: Open Visual Studio 2013 and create a new blank solution, name it as 'Modern_Charts_Service'. In this solution, add a new WEB API project. Call it as 'WebApi_Service'.

Step 2: To this project, add a new ADO.NET Entity Data model and select AdventureWorks2012 database. From this database select the *SalesTerritory* table for mapping. After completing the wizard, the designer will look like the following:

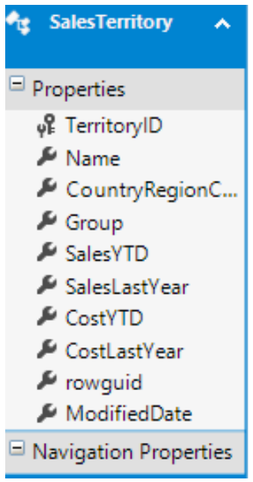


Figure 1: Sales Territory Table

In the application, we will using this table (as seen in Figure 1) to display chart by CountryRegionCode. For e.g. US, GB, etc.

Step 3: In the project, add a new WEB API controller project using Entity Framework:

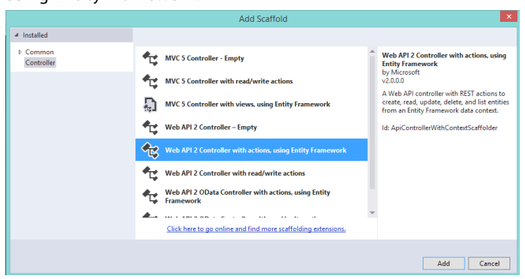


Figure 2: WebAPI 2 Controller project

Name the controller as 'SalesTerritoriesController'. This step will generate the code for data access from the database, using Entity Framework.

Now build the project.

Step 4: To the solution created in the first 3 steps, add a new WPF 4.5 application and call it 'WPF_ModernChart_Client'. In this project, add the assembly reference for the ModernUI Chart library for WPF. At the beginning of the article, we had discussed how to obtain this library. Once the library is added to your project, the project reference will look like the following in Solution Explorer:

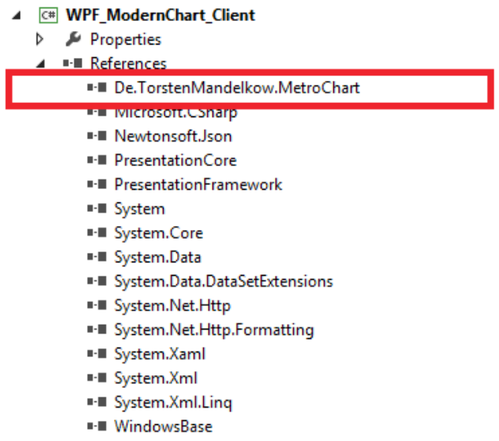


Figure 3: ModernUI Chart library

Step 5: In the project (bin\Debug folder) add an xml file with the name 'ChartNames.xml'. This file will store the different chart names in it:

```
<?xml version=”1.0” encoding=”utf-8” ?>
<ChartTypes>
  <Chart Name=”PieChart” Number=”0”></Chart>
  <Chart Name=”ClusteredColumnChart” Number=”1”>
  </Chart>
  <Chart Name=”ClusteredBarChart” Number=”2”></Chart>
  <Chart Name=”DoughnutChart” Number=”3”></Chart>
  <Chart Name=”StackedColumnChart” Number=”4”></Chart>
  <Chart Name=”StackedBarChart” Number=”5”></Chart>
  <Chart Name=”RadialGaugeChart” Number=”6”></Chart>
</ChartTypes>
```

We will be using this file to read the available chart types and display them on UI. The respective chart will be automatically drawn based upon the selected chart type.

Step 6: Add a new folder with the name 'HelperClasses' to the project. In this folder, add a new class file and declare some classes in it.

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

namespace WPF_ModernChart_Client.HelperClasses
{
  /// <summary>
  /// The class used to store the Chart Name and its
     number
  /// from the xml file.
  /// </summary>
  public class ChartNameStore
  {
    public int Number { get; set; }
    public string Name { get; set; }
  }

  /// <summary>
  /// The class used to load Chart types form the XML
     file.
  /// </summary>
  public class ChartTypeHelper
  {
    ObservableCollection<ChartNameStore> _ChartsNames;

    /// <summary>
    /// The Method Load the Xml file and return chart
       names.
    /// </summary>
    /// <returns></returns>
    public ObservableCollection<ChartNameStore>
    GetChartNames()
    {
      _ChartsNames = new
      ObservableCollection<ChartNameStore>();

      XDocument xDoc = XDocument.Load(“ChartNames.
      xml”);

      var Charts = from c in xDoc.
      Descendants(“ChartTypes”).Elements(“Chart”)
      select c;

      foreach (var item in Charts)
      {
        _ChartsNames.Add(new ChartNameStore()
        {
          Name =  item.Attribute(“Name”).Value,
          Number =Convert.ToInt32(item.
          Attribute(“Number”).Value)
        });
      }

      return _ChartsNames;

    }
  }
}
```

The *ChartNameStore* class is used for representing the chart information and the *ChartTypeHelper* class contains the *GetChartNames* method which loads and reads the xml file created in the previous step and returns all the Chart names.

Step 7: Since we are using WEB API, we need to add the ASP.NET WEB API client libraries from NuGet, into our WPF project. This step will add all the necessary libraries to make a call to WEB API.

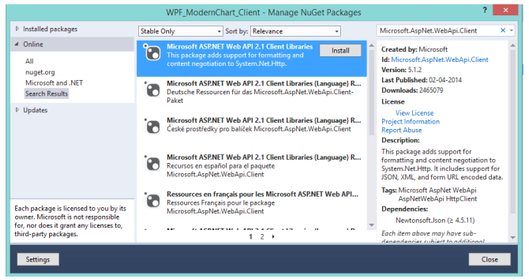


Figure 4: ModernUI Chart library

After installation, the WPF project will have the following references added as shown in Figure 5:

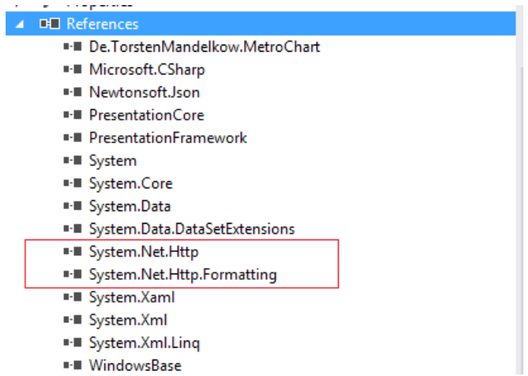


Figure 5: WPF Project references

Step 8: To the project, add a new folder with the name 'ServiceAdapter'. In this folder add a new class file and write the following code:

```
using System;
using System.Collections.ObjectModel;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;
using WPF_ModernChart_Client.ModelClasses;

namespace WPF_ModernChart_Client.ServiceAdapter
{
  /// <summary>
  /// class used to make call to WEB API and get
      the sales information
  /// </summary>
  public class ProxyAdapter
  {
    ObservableCollection<SalesTerritory> _SalesData;
    public async Task<ObservableCollection
    <SalesTerritory>> GetSalesInformation()
    {
      using (var webClient = new HttpClient())
      {
        webClient.BaseAddress = new Uri("http://
        localhost:6043/");
        webClient.DefaultRequestHeaders.Accept.Clear();
        webClient.DefaultRequestHeaders.Accept.Add(new
        MediaTypeWithQualityHeaderValue("application/
        json"));
        HttpResponseMessage resp = await webClient.
        GetAsync("api/SalesTerritories");

        if (resp.IsSuccessStatusCode)
        {
          _SalesData = await resp.Content.
          ReadAsAsync<ObservableCollection
          <SalesTerritory>>();
        }
      }
      return _SalesData;
    }
  }
}
```

This code makes use of the *HttpClient* class which helps to make an asynchronous call to WEB API. The *BaseAddress* property of the HttpClient class accepts the URI of the WEB API. The call is made over the uri and when completed, the data is read from it.

Step 9: To the project, add a new folder with the name 'ModelClasses' and add the following classes in it:

```
namespace WPF_ModernChart_Client.ModelClasses
{
  /// <summary>
  /// The entity class for Sales information
  /// </summary>
  public partial class SalesTerritory
  {
    public int TerritoryID { get; set; }
    public string Name { get; set; }
    public string CountryRegionCode { get; set; }
    public string Group { get; set; }
    public decimal SalesYTD { get; set; }
    public decimal SalesLastYear { get; set; }
    public decimal CostYTD { get; set; }
    public decimal CostLastYear { get; set; }
    public System.Guid rowguid { get; set; }
    public System.DateTime ModifiedDate { get; set; }
  }

  /// <summary>
  /// The class which is used to provide the necessary
      information to draw chart
  /// on the chart axis.
  /// </summary>
  public partial class SalesInfo
  {
    public string Name { get; set; }
    public decimal Sales { get; set; }
  }

  /// <summary>
  /// The class used to represent information for the
      CountryRegioCode
  /// e.g. US, GB etc.
  /// </summary>
  public class CountryRegionCode
  {
    public string CountryRegion { get; set; }
  }
}
```

In this code we just saw:

- The *SalesTerritory* class represents the entity for the sales information received from the WEB API.
- The *SalesInfo* class is used to provide the necessary information for drawing chart on the UI.
- The *CountryRegionCode* class is used to represent the CountryRegion information on the UI, so that when the end user makes a drop down selection, a chart can be drawn.

Step 10: In the project add a new folder with the name 'Commands' and add the following command class in it.

```
using System;
using System.Windows.Input;

namespace WPF_ModernChart_Client.Commands
{
  public class RelayCommand : ICommand
  {
    Action _handler;

    public RelayCommand(Action h)
    {
      _handler =  h;
    }
    public bool CanExecute(object parameter)
    {
      return true;
    }
    public event EventHandler CanExecuteChanged;

    public void Execute(object parameter)
    {
      _handler();
    }
  }
}
```

Step 11: To draw a Chart, some logic is required for reading and manipulating the received data from WEB API and then using it in a proper format to draw chart on the UI. Since we are using the *SalesTerritory* table to read the data, the data for columns for *SalesYTD* and *SalesLastYear* will be used for displaying Charts and the column *CountryRegionCode* will be used based upon which the data will be grouped to display on the chart.

Add a new folder with the name 'ViewModelsRepository' in the project and add a class file to it with the following code:

```
using System.Collections.ObjectModel;
using System.Linq;
using WPF_ModernChart_Client.HelperClasses;

using WPF_ModernChart_Client.ModelClasses;
using WPF_ModernChart_Client.ServiceAdapter;
using System.ComponentModel;
using WPF_ModernChart_Client.Commands;

namespace WPF_ModernChart_Client.ViewModelsRepository
{
  /// <summary>
  /// The class contains:
  /// 1. The ChartInfo collection property to display
      the available charts in the combobox.
  /// 2. The SalesData collection property to use as a
      source for drawing charts in UI.
  /// 3. The CountryRegion collection property to use
      as a source for the CountryRegion code for UI.
  /// 4. The CountryRegionName property to use as a
      filter when the UI changes the CountryRegion to
      draw chart.
  /// 5. The IsRadioButtonEnabled property for enabling
      and disabling the RadioButton on UI
  /// 6. The SalesDetailsYTDCommand and
```

```
    SalesDetailsLastYearCommand command object to
    execute methods when action is taken on UI
/// 7. The GetYTDSalesDataByCountryRegion() and
    GetLastYearSalesDataByCountryRegion() method are
/// used to get the YTD and LastYear sales data
    respectively based upon the CountryRegionCode
    selected on UI.
/// 8. The GetCountryRegionCodeGroup() method is
    used to provide information  about available
    CountryRegion code.
/// </summary>
public class ChartsViewModel : INotifyPropertyChanged
{
  ProxyAdapter adapter;

  ObservableCollection<ChartNameStore> _ChartsInfo;

  public ObservableCollection<ChartNameStore>
  ChartsInfo
  {
    get { return _ChartsInfo; }
    set { _ChartsInfo = value; }
  }

    ObservableCollection<SalesInfo> _SalesData;

    public ObservableCollection<SalesInfo> SalesData
    {
      get { return _SalesData; }
      set { _SalesData = value; }
    }

 /// Please refer to the code download to view the rest
     of the code

  }
}
```

The code we just saw has the following specifications:

- The ChartInfo collection property is used to display the available charts in the combobox.

- The SalesData collection property is used as a source for drawing charts in UI.

- The CountryRegion collection property is used as a source for the CountryRegion code for UI.

- The CountryRegionName property is used as a filter when the UI changes the CountryRegion to draw chart.

- The IsRadioButtonEnabled property for enabling and disabling the RadioButton on the UI

- The SalesDetailsYTDCommand and SalesDetailsLastYearCommand command object to execute methods when action is taken on UI

- The GetYTDSalesDataByCountryRegion() and GetLastYearSalesDataByCountryRegion () method are used to get the YTD and LastYear sales data respectively based upon the CountryRegionCode selected on UI.

- The GetCountryRegionCodeGroup () method is used to provide information about the available CountryRegion code.

Step 12: In the Project, add a new folder with the name 'ChartControls'. This folder will also contain some WPF UserControls. Each UserControl will contain the different chart types from the ModernUI Chart library. Add a new WPF User control in the project and name it as 'ClusteredBarChartUserControl'. Define the User Control as following:

```
<UserControl x:Class="WPF_ModernChart_Client.
ChartControls.ClusteredBarChartUserControl"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/
blend/2008"

xmlns:ModernChart="clr-namespace:De.TorstenMandelkow.
MetroChart;assembly=De.TorstenMandelkow.MetroChart"

mc:Ignorable="d" Height="636" Width="837">
  <Grid Margin="0,0,10,-37" Height="590" Width="840">
    <ModernChart:ClusteredBarChart
    ChartTitle="Sales Chart" ChartSubTitle="Territory
    wise sales Chart" Margin="25,0,21,46">
      <ModernChart:ClusteredBarChart.Series>
        <ModernChart:ChartSeries SeriesTitle="The
        Sales Chart" Height="580" Width="830"
        DisplayMember="Name" ValueMember="Sales"
        ItemsSource="{Binding}">
        </ModernChart:ChartSeries>
      </ModernChart:ClusteredBarChart.Series>
    </ModernChart:ClusteredBarChart>
  </Grid>
</UserControl>
```

In the XAML, the ModernUI Chart library is highlighted in yellow. The XAML defines property set for the 'ClusteredBarChart'. The ChartSeries defines the data source for drawing chart using the 'ItemsSource' property. The *DisplayMemberName* property displays the Textual data on the chart axis and *ValueMember* property is used to display values on chart axis.

Follow Step 12 to create user controls in a similar manner for the following chart types:

- PieChart
- ClusteredColumnChart
- ClusteredBarChart
- DoughnutChart
- StackedColumnChart
- StackedBarChart
- RadialGaugeChart

Step 13: Open MainWindow.xaml and add the following XAML markup in it:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
  presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/
  xaml"
  xmlns:ChartControls="clr-namespace:WPF_ModernChart_
  Client.ChartControls"
  x:Class="WPF_ModernChart_Client.MainWindow"
  Title="MainWindow" Height="856.982" Width="1472.565"
>
<Window.Resources>
  <DataTemplate x:Key="SaleTemplate">
    <TextBlock Text="{Binding CountryRegion}"/>
  </DataTemplate>
</Window.Resources>
<Grid DataContext="{Binding}">
  <Grid.RowDefinitions>
    <RowDefinition Height="22*"/>
    <RowDefinition Height="163*"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition Width="0*"/>
  </Grid.ColumnDefinitions>

  <TextBlock TextWrapping="Wrap" x:Name="txtname"
  TextAlignment="Center" Grid.Row="0" Grid.
  RowSpan="1" FontSize="30" FontWeight="Bold"><Run
  Text="Sales Chart for Territory-"/><Run Text="
  "/><InlineUIContainer>

  <TextBlock Height="47" TextWrapping="Wrap"
  Text="{Binding ElementName=lstCountryRegionCode,
  Path=SelectedItem.CountryRegion}" Width="212"/>
  </InlineUIContainer></TextBlock>

  <TextBlock HorizontalAlignment="Left"
  Height="39" Margin="10,156,0,0" Grid.Row="1"
  TextWrapping="Wrap" Text="Select Chart Type:"
  FontSize="20" VerticalAlignment="Top" Width="173"/>

  <ComboBox HorizontalAlignment="Left"
  Height="32" SelectedIndex="0"
  Margin="10,218,0,0" Grid.Row="1"
  VerticalAlignment="Top"
  x:Name="lstcharttype" Width="245"
  ItemsSource="{Binding ChartsInfo}"
  DisplayMemberPath="Name" SelectedValuePath="Number"
  RenderTransformOrigin="0.506,3.719"
  SelectionChanged="lstcharttype_SelectionChanged"/>

  <TextBlock HorizontalAlignment="Left"
  Height="31" Margin="10,15,0,0" Grid.Row="1"
  TextWrapping="Wrap" FontSize="20"
  VerticalAlignment="Top" Width="194">
  <Run Text="Select Country"/><Run Text=":"/>
  </TextBlock>
```

```
  <ComboBox x:Name="lstCountryRegionCode"
    SelectedIndex="0"
    HorizontalAlignment="Left" Margin="10,51,0,0"
    Grid.Row="1" VerticalAlignment="Top" Width="240"
    Height="30"
    ItemsSource="{Binding CountryRegion}"
    ItemTemplate="{StaticResource SaleTemplate}"
    SelectedItem="{Binding CountryRegionName,
    Mode=TwoWay,
    UpdateSourceTrigger=PropertyChanged}">
  </ComboBox>

  <Grid HorizontalAlignment="Left" Height="638"
    Margin="305,15,0,0" Grid.Row="1"
    DataContext="{Binding SalesData}"
    x:Name="grdChartContainer"
    VerticalAlignment="Top" Width="899">
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition Width="0*"/>
  </Grid.ColumnDefinitions>
</Grid>

  <RadioButton Content="Sales YTD"
  HorizontalAlignment="Left" Margin="10,108,0,0"
  Name="rdbtnsalesytd" Command="{Binding
  SalesDetailsYTDCommand}"
  IsEnabled="{Binding IsRadioButtonEnabled}"
  FontSize="20" Grid.Row="1" VerticalAlignment="Top"
  RenderTransformOrigin="0.337,1.311" Height="31"
  Width="131"/>

  <RadioButton Content="Sales Last Year"
  HorizontalAlignment="Left" Margin=" 157,108,0,0"
  Command="{Binding SalesDetailsLastYearCommand}"
  IsEnabled="{Binding IsRadioButtonEnabled}"
  FontSize="20"  Name="rdbtnsaleslastyear"
  Grid.Row="1" VerticalAlignment="Top"
  RenderTransformOrigin="0.337,1.311"
  Height="31" Width="188"/>

  </Grid>
</Window>
```

This XAML will produce the following output:

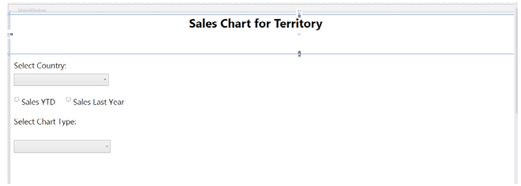


Figure 6: Sales Chart UI

The ComboBox 'lstcharttype' displays all the available chart types. The ComboBox 'lstCountryRegionCode' is used to display all the country regions in it so that end-user can select it to draw a chart. Both ComboBoxes are initially set to the default selected index '0'. The Grid with the name 'grdChartContainer' (not seen in the UI) will be used as a container to the UserControl for displaying chart types as defined in Step 12. The Grid is bound with the *SalesData* property declared in the ViewModel class, using *DataContext* property of the Grid. This property will then be passed to each user control for drawing chart. Both radio buttons are set with the Command property declared in the ViewModel class.

Step 14: Add the following code in the code-behind of the MainWindow.xaml:

```
public partial class MainWindow : Window
{
  ChartsViewModel vm;

  /// <summary>
  /// The Constructor set the DataContext of the Window
      to an object of the ViewModel class.
  /// </summary>
  public MainWindow()
  {
    InitializeComponent();
    vm = new ChartsViewModel();
    this.DataContext = vm;
  }

  /// <summary>
  /// The method gets executed when the chart is
      selected from the ComboBox.
  /// The method adds the UserControl in the Grid with
      the name grdChartContainer based upon the chart
      name selected from the ComboBox.
  /// </summary>
  /// <param name="sender"></param>
  /// <param name="e"></param>
  private void lstcharttype_SelectionChanged(object
  sender, SelectionChangedEventArgs e)
  {
    grdChartContainer.Children.Clear();

    var chartType = lstcharttype.SelectedItem as
    ChartNameStore;

    switch (chartType.Name)
    {
      case "PieChart":
        PieChartUserControl pie = new
        PieChartUserControl();
        pie.DataContext = grdChartContainer.
        DataContext;
        grdChartContainer.Children.Add(pie);
        break;

      case "ClusteredColumnChart":
        ClusteredColumnChartUserControl ccchart = new
        ClusteredColumnChartUserControl();
        ccchart.DataContext = grdChartContainer.
        DataContext;
        grdChartContainer.Children.Add(ccchart);
        break;

      case "ClusteredBarChart":
        ClusteredBarChartUserControl cbchart = new
        ClusteredBarChartUserControl();
        cbchart.DataContext = grdChartContainer.
        DataContext;
        grdChartContainer.Children.Add(cbchart);
        break;

      case "DoughnutChart":
        DoughnutChartUserControl dnchart = new
        DoughnutChartUserControl();
        dnchart.DataContext = grdChartContainer.
        DataContext;
        grdChartContainer.Children.Add(dnchart);
        break;

      case "StackedColumnChart":
        StackedColumnChartUserControl stcchart = new
        StackedColumnChartUserControl();
        stcchart.DataContext = grdChartContainer.
        DataContext;
        grdChartContainer.Children.Add(stcchart);
        break;
      case "StackedBarChart":
        StackedBarChartUserControl stbchart = new
        StackedBarChartUserControl();
        stbchart.DataContext = grdChartContainer.
        DataContext;
        grdChartContainer.Children.Add(stbchart);
        break;
      case "RadialGaugeChart":
        RadialGaugeChartUserControl rgchart = new
        RadialGaugeChartUserControl();
        rgchart.DataContext = grdChartContainer.
        DataContext;
        grdChartContainer.Children.Add(rgchart);
        break;
    }
  }
}
```

The code we just saw defines an object of the ViewModel class. This object is assigned to the *DataContext* property of the MainWindow so that all public properties defined in the ViewModel class can be bound with the UI in XAML. The *lstcharttype_SelectionChanged* method is executed when the chart is selected from the *lstcharttype* combobox. This method then adds the Chart UserControl in the grid *grdChartContainer*.

Step 15: Run the application and you should see the following:

Figure 7: Sales Chart for US

The default values for the Country Region is selected as 'US' (United States). The default Chart type selected is 'PieChart'. Select the Radio Button 'Sales YTD' or 'Sales Last Year' and the result will change as seen in Figure 8:

Pie Chart

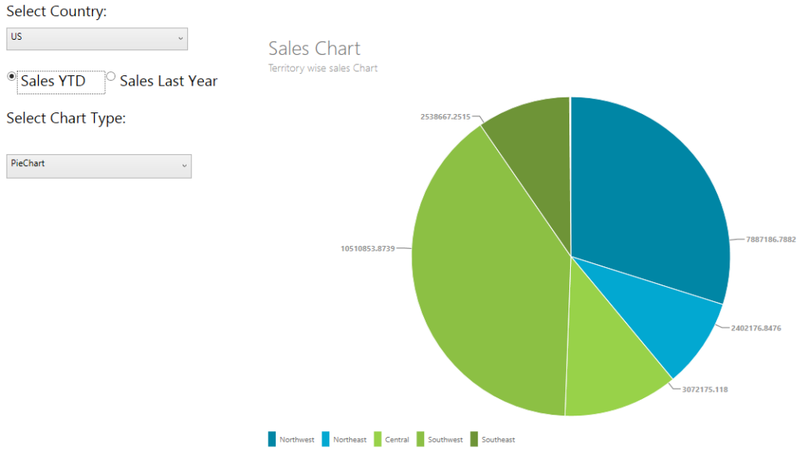


Figure 8: Sales YTD or Sales Last Year

Now change the chart type and see how the graphical representation changes:

Radial Gauge Chart

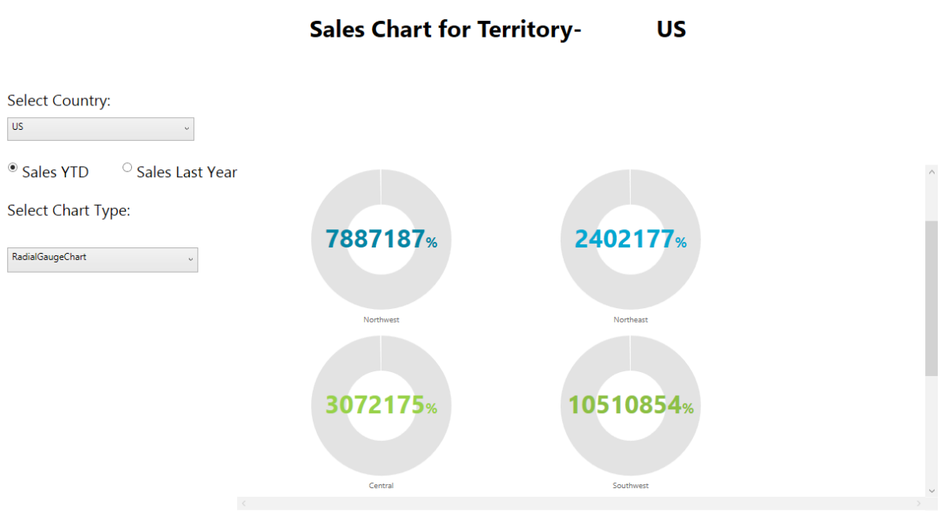


Figure 9: Radial Gauge Chart

Similarly all the other chart types can be tested by selecting the respective Chart type from the 'Select Chart Type' dropdown.

Clustered Column Chart

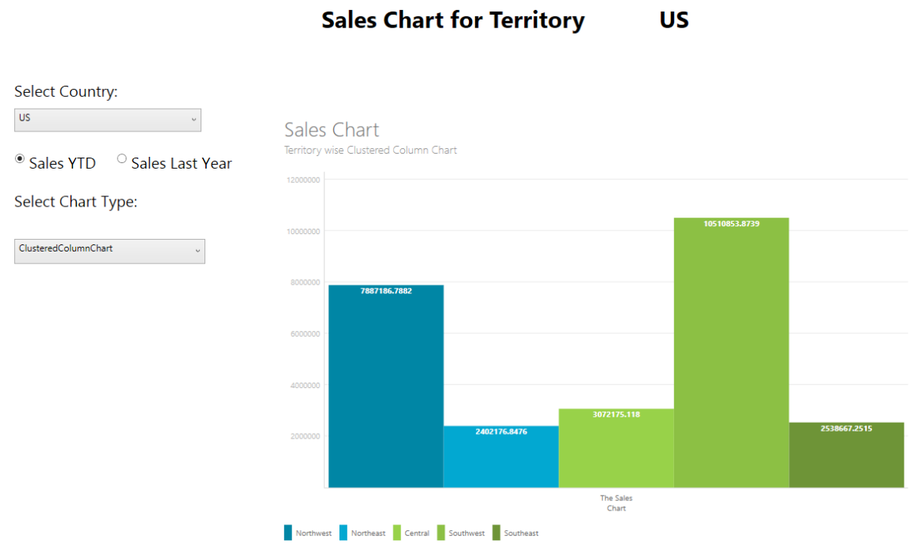


Figure 10: Clustered Column Chart

Clustered Bar Chart

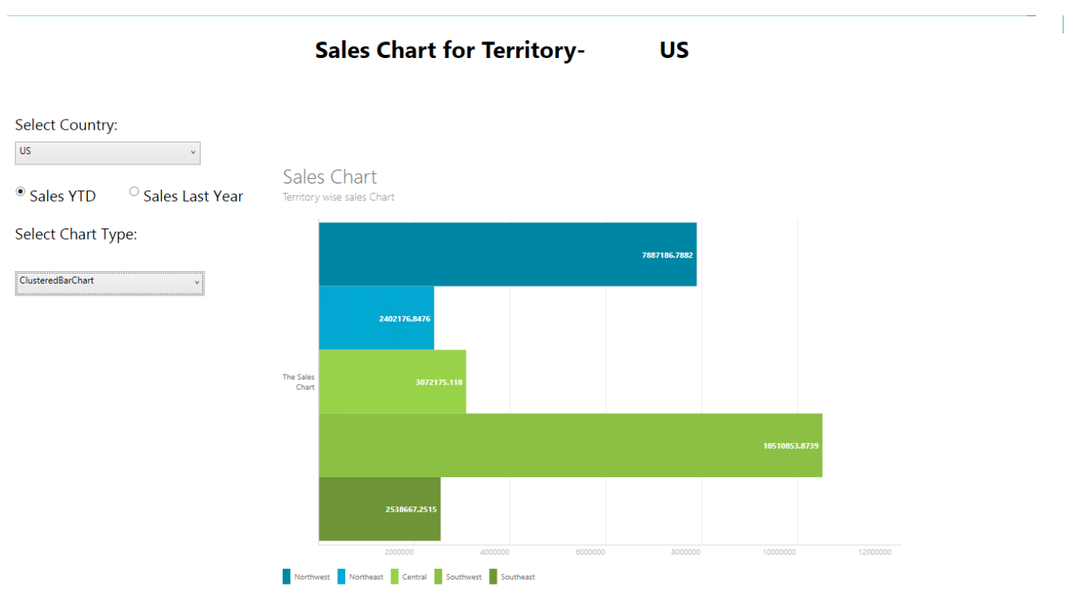


Figure 11: Clustered Bar Chart

Doughnut Chart

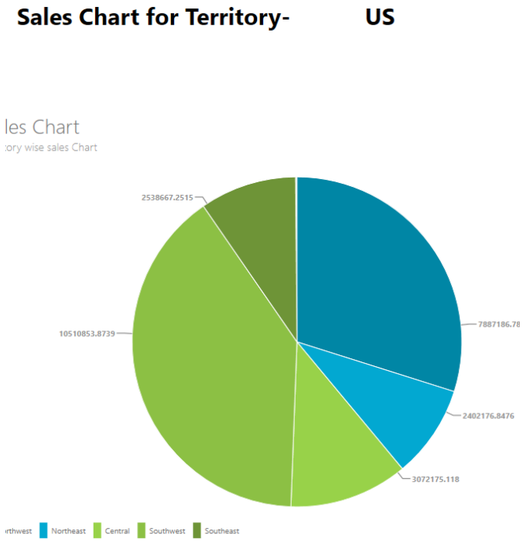


Figure 12: Doughtnut Chart

Stacked Column Chart

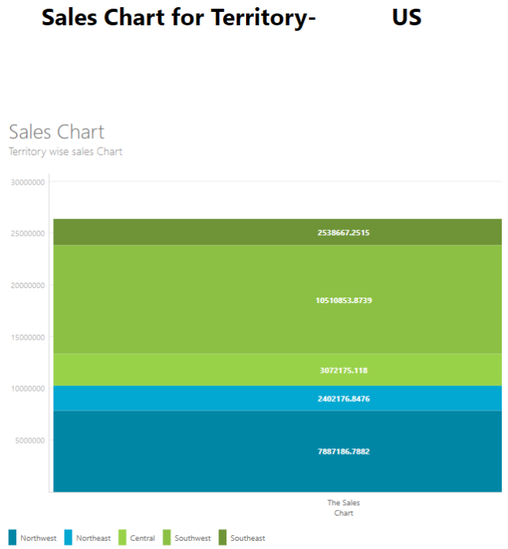


Figure 13: Stacked Column Chart

Stacked Bar Chart

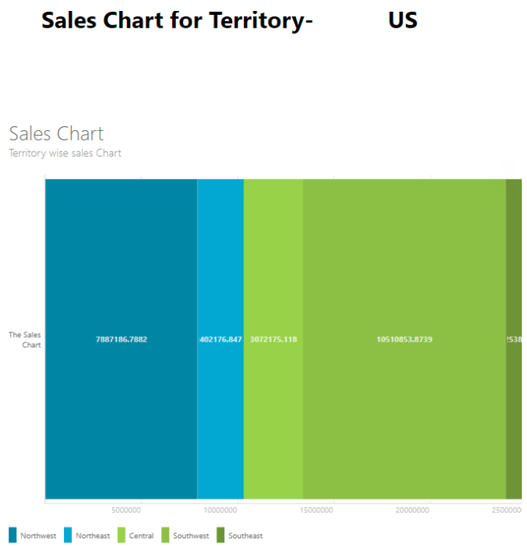


Figure 14: Stacked Bar Chart

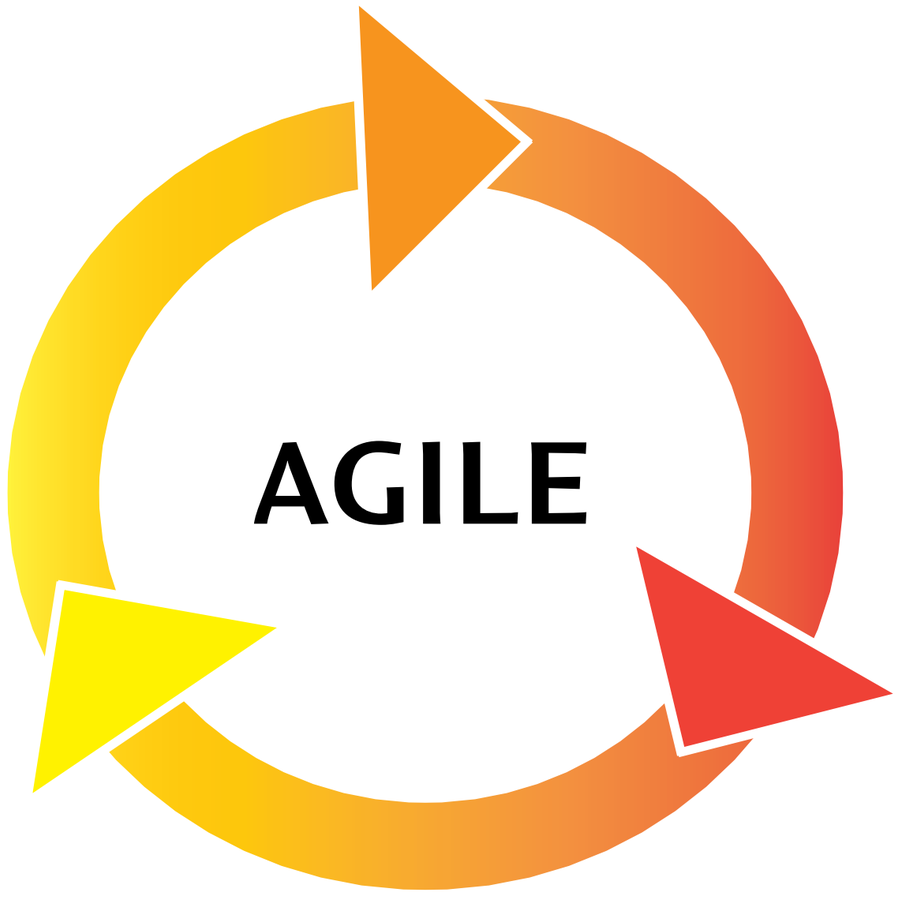
CONCLUSION

WPF has a rich charting and graphical capability, however it is time-consuming to build your own charts from groundup. The ModernUI chart library is a free handy library which allows you to quickly display some statistical data in a graphical form instead of creating your own chart using WPF 2-D APIs or relying on a 3rd party graphic package ■

Download the entire source code from our GitHub Repository at bit.ly/dncm13-modernuicharts

Mahesh Sabnis is a Microsoft Certified Trainer (MCT) since 2005 and has conducted various Corporate Training programs for .NET Technologies (all versions). Follow him on twitter @maheshdotnet. Mahesh blogs regularly on Azure, SharePoint, Metro UI, MVC and other .NET Technologies at bit.ly/HsS2on

SUPPORT FOR AGILE TEAMS IN VISUAL STUDIO 2013 AND TFS 2013



Agility is the ability to twist and turn in minimum time and efforts. A fighter aircraft is agile. It has to be, since it has to engage in dogfight with enemy fighter aircrafts and should be able to point its weapons towards the target, in minimum time. Fighter aircrafts are designed for stresses and strains due to agile movements. A passenger or cargo aircraft is not required to be agile. It does not need to change direction frequently and quickly. Once its direction is set, it can travel in that direction for hours together. If somebody tries to steer it too often and too fast, then the whole structure of that type of aircraft may fail, resulting in a catastrophic crash.

In the early days, software development was assumed to follow the pattern of passenger aircraft. The requirements of business were supposed to be fixed and changes if any, were not because business requirements changed, but because there was a gap in understanding the actual requirements. The process involved deciding the goals by knowing the requirements, set the direction of the team and start working in that direction. The project could go on for months without changing the original goal and the direction towards it. However during the subsequent years it became apparent that, by the time a significant number of requirements are implemented, business environment and needs of the business from the software, have also changed. Some of the requirements are no longer valid to the business. The team cannot change direction because it is not designed to be Agile. There are significant number of processes that it has to go through, before it can change direction. Eventually the Business that is sponsoring this software development, does not like it.

Since mid-90s the concept of agile teams was put forward by a few thinkers. They published the Agile Manifesto which provided the values and principles that guided the agile teams to work really agile. The main concept was to assume and embrace change. Agile teams should expect that business needs are bound to change as the business environment, which are dynamic, will change. Some other principals were to give importance to working software rather than documented requirements, technical excellence, short iterative approach that limits the plans to one iteration of maximum a few weeks etc. You can read all agile principals in the Agile Manifesto at http://agilemanifesto.org/principles.html.

To support the implementation of these agile principals, some practices evolved over a period of time. There are no fixed agile practices, they keep on growing as the experience of industry to make teams agile successful, is also growing. Some of these practices were codified for managing the agile teams and got formulated in the methodology like *Scrum*. Some other practices which were more technical in nature were given more importance in another methodology like *XP* (eXtream Programming). Number of such methodologies with preference to some practices to implement agile principals, have evolved in the past few years. An agile team is not restricted to follow only one methodology, but can make combination of multiple methodologies to suite their environment and interests. For example an agile team developing a new software may use Scrum for managing the team and planning, whereas for best effort development, it may follow XP.

Before any organization decides to enable agile teams, these organizations should understand that *agile is a very democratic way of working.* Agile teams are empowered to be self-organizing. They are also empowered to take decision to plan within an iteration. Let me extend the example of aircrafts to make my point. Until the WWII, aircrafts were piloted entirely by a human being. This person could control the components of the aircraft to make it twist and turn. The demand for agility increased to such an extent that limit of that human being was reached. Pilots could no longer control all the components of the fighter aircraft with the agility that was required. Slowly decisions at certain components were automated so that it may not harm the aircraft and keep it flying without intervention from the pilot. A new technology called fly-by-wire came into being. In this technology, components of aircraft had computers that took decisions based upon the environment. Aircraft controlled itself so that it remained the Agile. The Pilot became only a facilitator for the computers to take appropriate decisions. Similar to that, organizations should *empower* agile teams to take their own decisions so that they remain most agile and receptive to any changes in requirements. The role of Project Manager should evolve into a *facilitator* that enables agile team to take correct decisions. I have seen many organizations across the continents (and their customers) that intend to build agile teams, but *do not empower* the team to take any decision and continue to have a hierarchical decision structure. This drastically constrains the agility of the team.

If the organization is willing to make Agile teams self-organizing and to empower them for decision making, then

there are a plethora of tools that will help agile teams to be agile. Managing the agile team is to decide which roles should be present, what are responsibilities of each role, activities that every team member does and timeline of these activities. Scrum provides the framework and guidelines for these decisions.
To know more about Scrum, read the guidance provided at https://www.scrum.org/Scrum-Guide.

Team Foundation Server (TFS) 2013 facilitates implementation of Scrum by providing a process template and a set of tools built in it.

Roles specified in Scrum are Product Owner, Scrum Master and Team Members. We can create groups in TFS for each of these roles. Although it is not of any significance right now, we may give specific access rights to each group when required. TFS 2013 supports Scrum by providing tools for creation of Product Backlog, Sprint Planning, and Team Capacity Balancing etc. Product backlog is created by adding required number of work items of the type "Product Backlog Item" (PBI). Entire Product Backlog can be viewed either by executing a pre-built query "Product Backlog" or by going to the screen of Visual Studio Online > Team Project Name > WORK > Backlogs > Backlog Items. I am going to stick to the second approach as for a couple of other features, it is the only approach that is available. The screenshot in Figure 1 shows the entire product backlog at a specific moment in time. Product Backlog is a living document that grows as new requirements are identified and added by the Product Owner. Items in product backlog are prioritized by the product owner. This becomes useful for the team to select PBIs for the sprint.

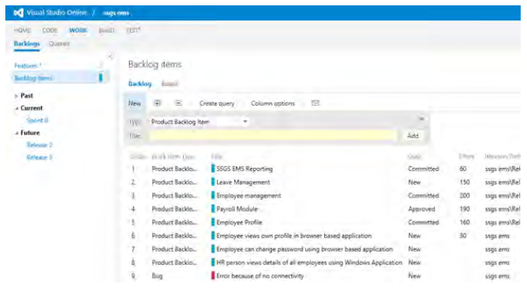


Figure 1: Backlog items as seen in Visual Studio Online

One of the important concepts before the PBI is finalized, is the validation of PBI by the customer. Product owner takes help of the team to create a storyboard pertaining to the screens that will be created to implement PBI. These screens are just the mock-up of the UI. TFS 2013 and MS PowerPoint collaborate to provide a storyboarding tool to the team and the customer. When a PBI is opened in the Web Access, there is a separate tab for StoryBoards. In that tab, the team member can create a storyboard for that PBI.

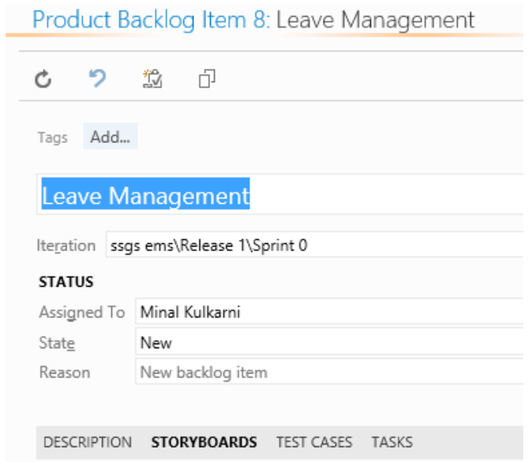


Figure 2: Backlog items as seen in Visual Studio Online

When the link to Start storyboarding is clicked, it opens the installed storyboarding tool that is embedded in MS Project. It allows the team member to create the screen mockups. Every slide contains a mockup of one of the screens that will be built.

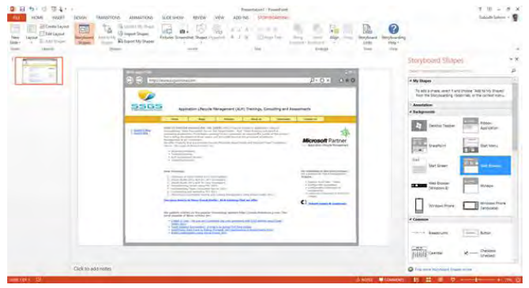


Figure 3: Storyboarding tool in MS Project

There are shapes provided in PowerPoint to specially create the mockups of the various types of applications that can be created in Visual Studio 2013.

PBI with its linked storyboard is assigned to Product Owner or a customer representative. They can open the storyboard to give comments on it and save it back. After that, they can reassign it back to the team member. In this way the functionality of PBI is validated before that PBI is set to *Approved* state. Finalized product backlog can be viewed in the TFS Web Access and may look like this:

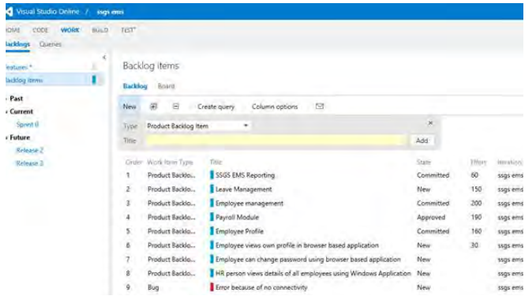


Figure 4: Product Backlog as viewed in TFS Web Access

Status of all PBIs of the product backlog can be seen from the *Board* view of the backlog.

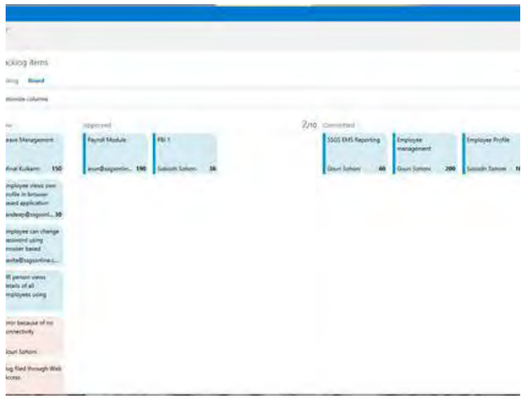


Figure 5: Status of Product Backlog PBIs

Scrum process template built into TFS 2013 facilitates iterative process by allowing us to create Sprints in the iterations framework. These sprints can be of the length decided by the team or the organization. Team can set the start and end date for the sprint that is added. In the Backlogs view, the sprint that spans the current date is shown as Current sprint.

Before the team can decide on the list of PBIs for a sprint, it should have an estimate of available number of hours for the sprint. There may be some team members who work on multiple projects and cannot devote a full day, but only a certain number of hours per day for this project. It is also possible that some of them are not available for a few days in between. Such details will be used for sprint planning and can be entered in the *Capacity* screen of each sprint.

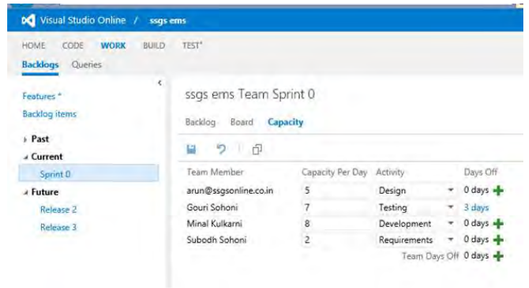


Figure 6: Capacity screen for each sprint

Team can move a PBI to a sprint by using drag and drop on the sprint name in this view itself. Each PBI can have a value for *Effort* which is a relative value of efforts estimation. It may or may not be the estimate of actual hours required to make that PBI 'Done'. It also has a field for *Remaining Work*. All the *Remaining* Hours of the children tasks are rolled up in this field automatically.

Some of the PBIs may not have all the details. That is what is expected by an Agile team. It keeps on gathering data and analyzing it even after it has started working on other PBIs in the sprint. Unlike traditional way, all the requirements are not frozen before the work starts. This gives the team required space to maneuver. Backlog view of the current iteration may look like this:

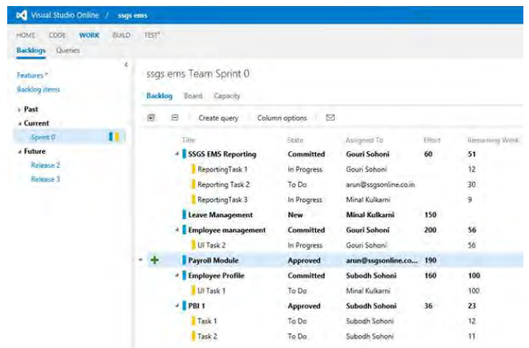


Figure 7: Current Iteration Backlog

On the right side of the screen (shown in Figure 8), graphs are shown for Capacity vs. Estimated Efforts.

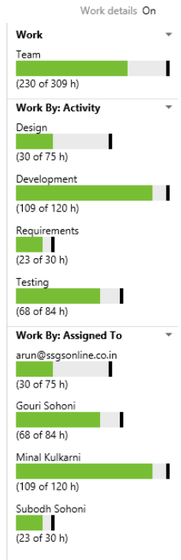


Figure 8: Capacity vs Estimate Effort

The image shows that the team and its team members are not being utilized to full capacity. This is natural since details of some of the PBIs are yet to be entered. Ideal situation will be when capacity of team and every team member is nearly 100% utilized. Team can put some PBIs back on the product Backlog and bring other PBIs in on the Sprint Backlog.

When the team starts working on the PBIs and their children tasks, the state of those work items is changed from *To Do - In Progress – Done*. Scrum provides a graphical display of these PBIs and Tasks in various state in a snapshot at a time as Scrum Board or Task Board.

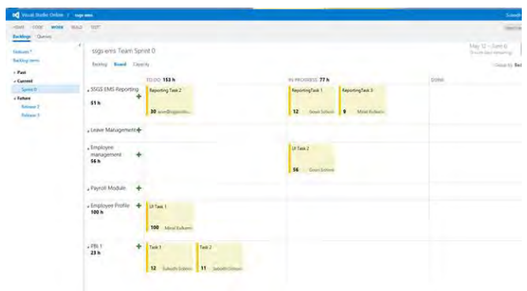


Figure 9: Scrum Board or Task Board

As the sprint progresses, the tasks are closed and PBIs are done. The Remaining work for that sprint reduces and at the end of that sprint, it should become 0. The trend of this reducing work is graphically shown as *Burndown* Chart, as seen in Figure 10.

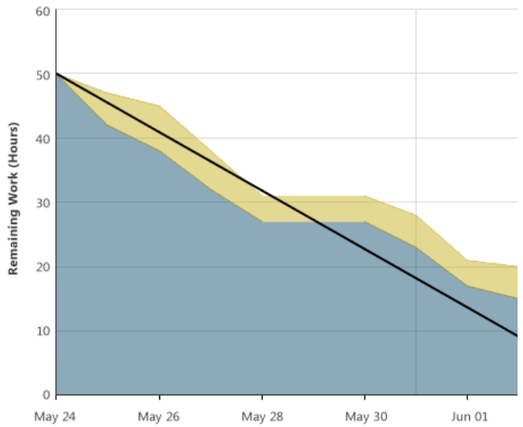


Figure 10: Burndown Chart

When the entire sprint is over, the 'Done' application is hosted on a server. A feedback request is created.

AGILE DEVELOPMENT PRACTICES

Now that we have seen how the Agile team management works for planning and running sprints, let us focus our attention to agile development practices. Some of these development practices are part of the eXtreme Programming (XP) but we are not limited by that. We are going to visit some of those practices which can improve the quality of software that is being developed, without affecting the productivity of the team adversely.

Test Driven Development (TDD)

The first practice that I would like to take up is *Test Driven Development* (TDD). Underlying to this practice is the concept that every bit of code that is written, should be unit tested. For clarity, the unit testing is to create code that executes the application code with parameters, for a known output. Such code once created can be executed after any change in the code of the method under test to ensure that it's functional integrity has not got affected. There are number of frameworks that allow you to run such unit tests. Microsoft Visual Studio 2013 supports many of such unit testing frameworks like NUnit, xUnit and off course Microsoft Unit Test. It has also introduced concepts of Fakes and Shims for easier testing of methods that expect custom objects as parameters. Visual Studio 2013 also supports Code Coverage computation which is the percentage of lines of the targeted code that is being tested by all the unit tests. If this percentage is not high enough, then there is a chance that untested lines of code may contain some bug.

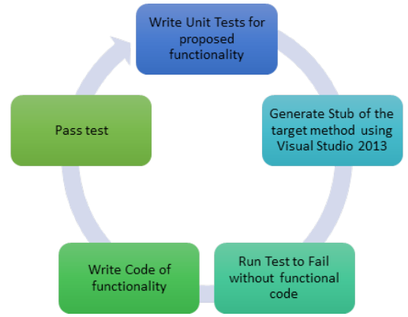


Figure 11: Test Driven Development

A twist in the tail of TDD unit testing is to write it first, even before the code to be tested is written. The whole idea is that you keep on coding the target method, just enough to pass the unit tests, that already are written. If all your unit tests comprehensively represent all the requirements from that method, then the method code that just passes all the unit tests, is the most optimum and lean without any flab of superfluous code. It can improve the productivity of development. Visual Studio allows you to write such test and then generate the stub of the method to be tested. This improves the productivity of the Agile Team further.

Code Refactoring

Refactoring the code is another agile development practice that teams use to make code optimum. Visual Studio 2013 provides the following features that supports refactoring:

1. **Rename** – Name of any entity like class, method, field, property etc. can be changed and wherever it is needed like calling methods or field references, the change is propagated.

2. **Encapsulate Field** – A field can be converted to a property.

3. **Extract Method** – Code in a method that is required repeatedly or is too large for comfort, can be converted to a method. Where the code originally existed, there the call to newly created method is inserted.

4. **Extract Interface** – From an existing class, we can generate an interface containing selected members of existing class.

5. **Remove Parameter** – Easy way to remove a parameter from a method. Wherever the method is called, the same parameter from the call will also be removed.

6. **Reorder Parameter** – Easy way to reorder the parameters of a method. Wherever the method is called, the reordering of the parameters will take effect.

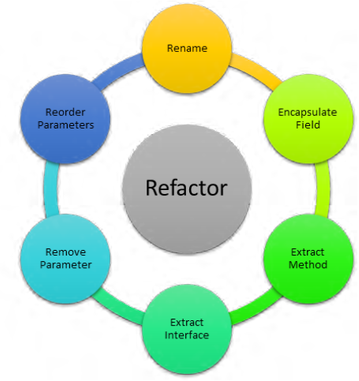


Figure 12: Refactoring the Code

Continuous Integration

Another very common practice of Agile Development is *Continuous Integration*. As soon as a functionality is developed by a team member, it should be checked that it integrates with the functionality of other features and does not break any of the previously written code. This is achieved by executing a server side build as soon as the new feature is accepted by the customer (can be during a demo).

Build service of TFS supports many types of triggers. One of them is to run a build as soon as someone checks-in code in the targeted project. That build can have standard workflow activities to compile code, execute code analysis on that, run configured unit tests and may also have custom activities to deploy the application. TFS 2013 encapsulates the entire

process of build and deployment through the feature for Release Management that is introduced recently.

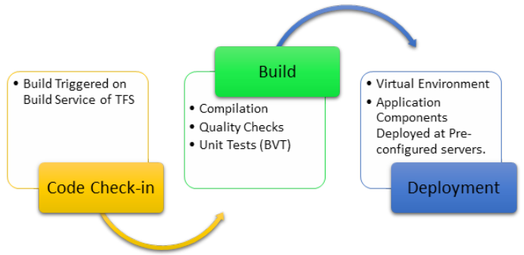


Figure 13: Continuous Integration

CONCLUSION

Agile is a way teams think, behave and work. Teams that are empowered to take decisions and implement those decisions based upon the changes in the business environment, become Agile. They get an edge over other teams as they accept there are changes in the needs of business and are ready to face those changes.

Team Foundation Server 2013 and Visual Studio 2013 are designed to help teams to become Agile. In this article, I explained how these software's facilitate implementation of Agile practices for management and for development ■

Subodh Sohoni, is a VS ALM MVP and a Microsoft Certified Trainer since 2004. Follow him on twitter @subodhsohoni and check out his articles on TFS and VS ALM at http://bit.ly/Ns9TNU

SOFTWARE GARDENING

STORING SEEDS

I have given you the basics of Software Gardening in my previous columns. We have visited topics such as: why construction is not a good metaphor as well as talked about soil, water, and light. Now, it's time to begin looking at specific techniques, tools, and practices that you use often, perhaps every day, in Software Gardening.

Gardeners and farmers often store seeds from one season to the next. But they need to be stored properly or they'll not be good the next season. The seeds may not germinate or the crops won't be hardy.

> *In Software Gardening, storing seeds is the practice of Version Control. But you must store the code correctly and use it correctly. If not, your harvest will not be good. In software terms, the Release will not go well.*

Unfortunately, I have found after years of looking at version control practices and talking with hundreds of developers, very few shops do version control correctly, handling branching and merging in a way that makes harvesting the application difficult.

BRANCHING PRACTICES

Before looking at branching and merging techniques, let's cover some basics. First, no matter the size of your team, one person or many, you should be using some type of version control.

Second, notice *I don't call it source control*. This is on purpose. Everything, not just source code, belongs in version control. Design documents, icons, notes, etc…anything and everything should be placed in version control.

Third, make sure you're using a good version control system. If you're still using Source Safe, I strongly encourage you to move to something else. Now! Visual Studio supports both TFS and Git right out of the box, both are good choices. Subversion is another good choice and integrates into Visual Studio through a third-party product. There are several other version control products that do a great job.

Finally, the frequency of check-in is important. It should be often, even several times a day. Once you get code working AND passing unit tests, it's time to check-in. Do not wait for days and days. This actually makes things worse, as we'll see in a moment. I break down each user story into tasks. When a task is complete, I check in.

One of the primary functions of version control is the ability to branch the code and then merge it again. There are several reasons why a team would branch the code. The first is physical separation of files, components, and subsystems. A branch would be made for each item that is physically separated from another.

Functional separation is another reason for branching. This includes features, logical changes, bug fixes, enhancements, patches, or releases.

Branching may also be done for environmental differences. For example, a change could be made to the build or runtime environment. A new compiler could be available. The application may use a new windowing system, third-party libraries, hardware, or operating systems.

How the team is organized may also be a reason for branching. It could be the activities, tasks, or subprojects needed to get the work done, are different enough, that the team wants to branch. Or it could be how the team itself is organized into roles and groups, with each group wanting a different branch.

Lastly, a team may branch based on procedural needs. The team's work behavior, company policies, processes, or procedures may cause branching.

Most of these are the wrong reasons for branching. In fact, when you properly implement Continuous Integration (I will discuss CI in a future column), branching is an anti-pattern. Let's look at a common branching scenario (Figure 1), one that may be similar to how you branch on your team. (The reality is, the branching practices of many companies are more complex than this example)

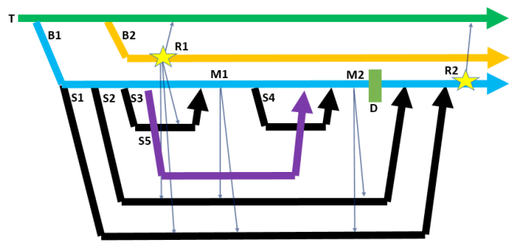


Figure 1. A common branching scenario

In Figure 1, the line T is the trunk. B1 is a branch made to work on new features. Off of B1 are several branches, S1 to S4. Each is branched off of B1 and at some point in the future, merged back into the B1. But at some point, a critical bug is found in the released code, so branch B2 is made to fix the bug. Once fixed, it is released (R1). The fix must then be merged into the trunk, branch B1, and all its sub-branches that are currently in development (the lines coming out of R1). Now imagine if there is some sub-branch S5, that begins after S3 but finishes after S4 begins, but before it ends. Its changes may also need to merge into S1 and S2 (those lines have been left off to simplify the diagram).

Finally, the pre-defined release date, D is reached, so merge of S1 and S2 into B1 must happen. This means code that has been branched for a long time, and has lots of changes that must merge. Not only once, but twice, one for each sub-branch. Once the code for B1 is released, it must be merged into the trunk. Now imagine what a mess would exist if there were branches B3, B4, B5, etc, that must also be merged into the trunk, at the same time as B1. Note that B1 and B2 end with an arrow, indicating they never end. You maintain them for an unspecified period of time. This is an ugly mess and causes delayed releases, fewer features, and lower quality. There must be a better way.

WORK ON THE TRUNK

The fact is, there is a better way. Always work on the trunk. At first thought, it sounds like this would create more problems, but in fact, it frees the team. The last minute merging doesn't happen because the code has been merged AND TESTED, regularly along the way. The key caveat is the trunk must always and ALWAYS, be releasable. There are four ways to accomplish always working on the trunk: hide new functionality, incremental changes, branch by abstraction, and components.

HIDE NEW FUNCTIONALITY

Sometimes there are new features that you need to add that take so long they can't be accomplished in a single release. Teams are tempted to branch, work on the branch over one, two, or more releases; then merge in when completed. This is what causes major merge issues. The correct way to handle this is to hide the new functionality.

The way you do this is to work on the branch, but turn the new functionality on and off through configuration settings. This makes the features inaccessible to users until the feature is complete. When the feature is done, remove the configuration settings. This way of working makes planning and delivery easier. Because you work entirely on the trunk, there is no branching, so your version control looks like Figure 2.

Figure 2: Working on the trunk eliminates awkward branching.

INCREMENTAL CHANGES

This is another technique to use when you have large changes to make and can be used in conjunction with function hiding. With incremental changes, you break down major changes into small parts and implement each on the trunk. Again, your version control looks like Figure 2.

BRANCH BY ABSTRACTION

The next technique sounds like you make a branch based on the name Branch by Abstraction, but it isn't. This technique consists of six steps.

1. Create an abstraction over the code that needs to be changed.

2. Refactor the code to use the abstraction

3. Create the new implementation

4. Update the abstraction to use the new code

5. Remove the old code

6. Remove the abstraction layer if it's not needed

You still work on the trunk, so it still looks like Figure 2.

COMPONENTS

The last way to work on the trunk is through the use of components. This technique is used in several different conditions:

- Part of the code needs to be deployed separately
- You need to move from a monolithic codebase to a core and plugins.
- If you need to provide an interface to another system
- Compile and link cycles are too long
- The current code is so large and complex, it takes too long to open in the IDE
- The codebase is too large for a single team

Components are developed independently of each other. One component references compiled code of another and does not use the code itself. Version control will consist of several trunks, one for each component (see Figure 3).

Figure 3. Working with components

While components solve some problems, you have to be careful as they can cause some problems too. First, your application has components everywhere. Everything can become a component. Second, beware of God components that do everything or control everything. Third, you may be tempted to have a team responsible for one or more components. Instead, a team should be responsible for a piece of functionality. Finally, having lots of components increases dependency management. This can be reduced through good, automated packaging and deployment systems.

SMART BRANCHING

With all this talk about working on the trunk, you may think that's the only option for handling code changes. There are still times when you may need to branch, but you need to do this in a way and time that make sense. I call this "Smart Branching" and there are three conditions where Smart Branching makes sense: branch for release, branch by feature, and by team.

BRANCH FOR RELEASE

It's important to branch to indicate in your version control system where a release happens. This is one place where branching makes sense. As explained earlier, you still develop on the trunk. When the code is ready for release, create a branch and release from there. Critical defects that are found after release are committed on the branch, then merged into the trunk.

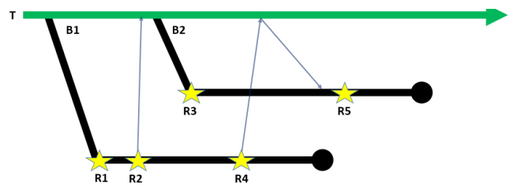


Figure 4. Branch for release

In Figure 4, the code is ready and branched as B1, then release (R1). At some point, a critical bug is found, fixed, and a new release is made (R2), then the code merged is into the trunk. The next release is then ready, so the code is branched (B2) for that release (R3). Another critical bug is fixed in B1, then R4 releases code to fix it. That fix is merged into the trunk, then down to B2. It then needs to be released R5 so customers have that fix.

One important note is that neither branch continues forever. At some point, the team determines that no more work will be done and the branch is ended. This is indicated by the bubble at the end of each branch line.

Some teams, as an alternative to branching at the time of release, opt to tag the trunk, then go back to that tag and branch when a critical bug is fixed. There is some discussion around this practice and opinion seems to be split about if this is a good practice or not.

BRANCH BY FEATURE

If you simply can't work on the trunk, then you should branch by feature. In this scenario, each user story is a branch. The number of branches equals the number of user stories the team is currently working on. But no branch should be active more than a few days.

Testing is performed on the branch. When it passes QA, it

is merged into the trunk. Changes to the trunk should be merged daily into active branches. Any refactorings are merged immediately.

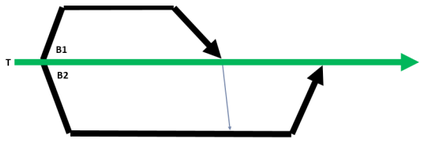


Figure 5. Branch by feature

Looking at Figure 5, branches B1 and B2 are made for two user stories. B1 is completed and is merged into the trunk, then merged again into B2.

BRANCH BY TEAM

Branch by team is similar to branch by feature (see Figure 6). Remember that earlier I said if you have multiple teams, each team works on a feature. The primary difference is that branches are merged into the trunk, then immediately into other branches when they are stable rather than after passing QA.

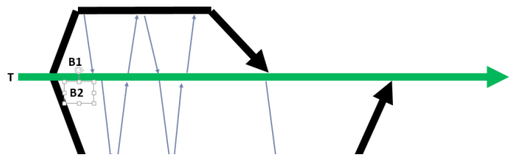


Figure 6. Branch by team

One key to making this type of branching work is the teams must be small and independent. Large teams cause problems as there gets to be so many changes to the branch, that it becomes difficult to merge into the trunk.

BEST PRACTICES

Before finishing up, here are some additional best practices to follow

• Compare before you commit. Make sure you have the latest changes and they don't break your code.

• Build and test before every commit. Don't break the build by checking in bad code.

• Build and test after every merge. Make sure you didn't break someone else's code.

• Explain commits. Add check-in comments so you know what's in that change.

• Read merge comments from other developers. This keeps you up-to-date with other things happening in the project and may alert you to possible conflicts with your code.

• Group commits logically. The changes in each commit should be related to each other.

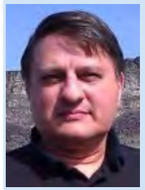
• Only store what's manually created. In other words, don't commit binaries. Especially ones that are generated from your code.

• Don't obliterate. In other words, don't just delete code without adding comments about what you did.

• Don't comment out code. How many times have you seen code with dozens of commented lines. This is unnecessary. If you're following good version control practices, you'll be able to go back and find those lines if you ever need them back.

SUMMARY

Hopefully now you'll be able to improve your branching and merging techniques and in turn improve your code. The two key things you should remember are to work on the trunk and check-in often. By following the practices here, your Software will grow and be lush, green, and vibrant ■



Craig Berntson is the Chief Software Gardener at Mojo Software Worx, a consultancy that specializes in helping teams get better. He has spoken at developer events across the US, Canada, and Europe for over 20 years. He is the co-author of "Continuous Integration in .NET" available from Manning. Craig has been a Microsoft MVP since 1996. Email: craig@mojosoftwareworx.com, Blog: www.craigberntson.com/blog, Twitter: @craigber. Craig lives in Salt Lake City, Utah.

Ryan Dahl introduced Node.js in his talk at JSConf 2009 and his talk received a standing ovation from the audience. Node.js is not a JavaScript library; it is a platform for creating web applications on the server side, using JavaScript. Built with C++ and JavaScript, Node.js uses V8 (JavaScript engine that comes with Chrome) as its engine to process JavaScript on the server. Using Node.js, one can quickly build and run a web application within a few minutes. Node.js works the same way as JavaScript works on any modern browser.

Node.js is event based and asynchronous. It assumes that all I/O operations are slow. So, they are all executed asynchronously. This makes execution of server, non-blocking.

Node executes every operation as an Event. Events are executed in an event loop. The server has only one thread to execute all the operations. Event loop keeps listening to the server events. When it encounters an asynchronous event like fetching data from a database, the operation is performed outside the event loop. Event loop then frees up the thread and responds to any other event that comes in. Once the asynchronous operation (here assuming it is fetching data from a database) completes, the Event loop receives a message about its completion and the result is processed once the event loop is free.

The pipeline of Node.js is very light weight. Unlike other server technologies, Node.js doesn't have a lot of components installed by default. It has just the right number of components to start the server. Other components can be added as they are needed.

BUILDING NODE.JS APPLICATIONS IN VISUAL STUDIO

PRE-REQUISITES:

In this article, we will build a Movie list application using Node.js, Express.js, Mongo DB and Angular JS on Visual Studio. I have used Windows 7 64-bit and Visual Studio 2013 Ultimate to develop the sample. You can use any OS that supports Visual Studio 2012 or 2013. Node Tools for Visual Studio can be used with Visual Studio 2012 Web Developer Express, Visual Studio 2013 Web Developer Express, or any paid versions of Visual Studio 2012 or 2013 (Professional, Premium or Ultimate).

As for the pre-requisites, you must have the following tools installed on your system:

1. Visual Studio 2012 or 2013
2. Node.js
3. NodeJS tools for Visual Studio (NTVS)
4. Mongo DB

The first three installations are straight forward. You just need to download and install the tools. For Mongo DB, you will get a zip file containing a set of binaries. Extract the zip to any location. I prefer it to be inside D:\MongoDB\bin. Once extracted, follow these instructions:

- In the folder named MongoDB, create a folder named 'data' and inside this folder, create another folder and name it 'db'

- Now go to the D:\MongoDB\bin folder and create a text file. Change name of the text file as 'mongod.conf' and paste the following statements in it:

```
dbpath = D:\MongoDB\data\db
logpath = D:\MongoDB\mongo-logs.log
logappend = true
```

Feel free to modify *dbpath* and *logpath* as per the path in your system.

- Run the command prompt as 'Run as administrator', change path to the bin folder of MongoDB on your system and run the following commands:

```
mongod.exe -f mongod.conf –service
net start mongodb
```

Once these commands are executed, MongoDB is up and running on your system as a service.

BUILDING A MOVIE LIST APPLICATION

Open Visual Studio 2012 or 2013 and choose File > New > Project. If you have installed Node tools for Visual Studio, you must be able to see the Node.js option under JavaScript and a number of project templates for creating Node.js application as shown in Figure 1:

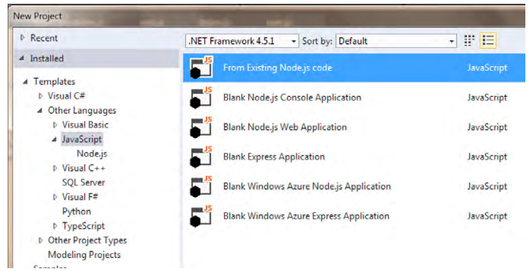


Figure 1: Node.js templates in Visual Studio

WHAT'S IN THE TEMPLATE?

From the list, choose *Blank Node.js Web Application* and change the name to 'NodeMovieList'. Once the project is created, observe the structure of the project in the Solution Explorer. You will find the following items:

- npm: Displays the node.js packages installed in the project. As we started with an empty project, it doesn't show anything as of now

- package.json: This file contains the list of Node.js packages and their versions needed by the project. It is similar to packages.config file used with NuGet

- server.js: This file is the starting point for the Node.js application. It defines the port on which the application has to run, contains calls to any middle wares required and defines actions for routes

If you check the server.js file, it has a very simple response defined. It sends Hello World in response to any incoming request. Run the application now and you will see a "Hello World" on your browser. When the application is running, you will see a console popping up. It is not a console created by

Visual Studio, it is the console installed with Node.js.

Debugging server.js in Visual Studio

We can place a breakpoint in the server.js file and debug the file just as we debug C# code files.

Figure 2: Debugging server.js in Visual Studio

All variables and objects in the code can also be inspected as seen in Figure 3:

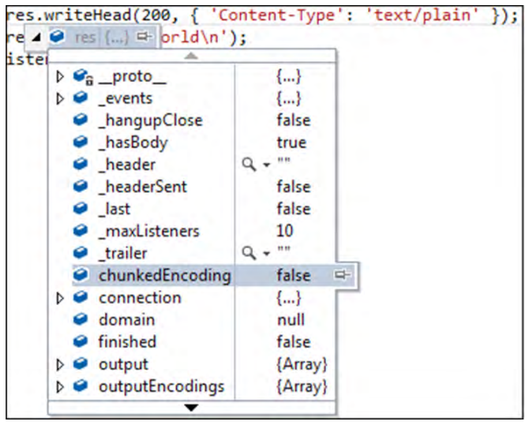


Figure 3: Viewing contents of an object

Our favourite Visual Studio debugging window namely Locals, Immediate Window, Call Stack and Watch window can also be used:

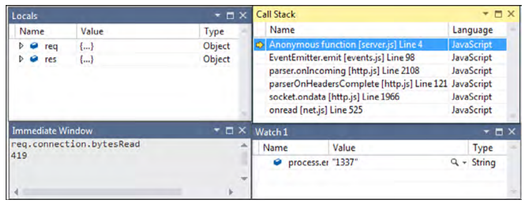


Figure 4: Visual Studio Debugging windows

GETTING EXPRESS.JS

The code in the server.js file doesn't do anything more than just starting a server on a specified port and writing some plain text to it. One can follow the same technique to respond with HTML, JSON or any type of data. But, this is done at a very low level. To write more complex applications, we need an abstraction layer that would make our job easier. To do so, we have a popular server framework called Express.js. Express has a number of middle wares defined that can be added to the Node.js pipeline to build robust applications. Express.js can be installed in the project using NPM.

NPM packages can be installed using either command line or using the GUI provided by Visual Studio. Node.js tools are smart enough to detect any package installed in the application, irrespective of the way you chose to install and list them under the NPM node in the Solution Explorer.

Right click on NPM and choose *Manage NPM Packages*.

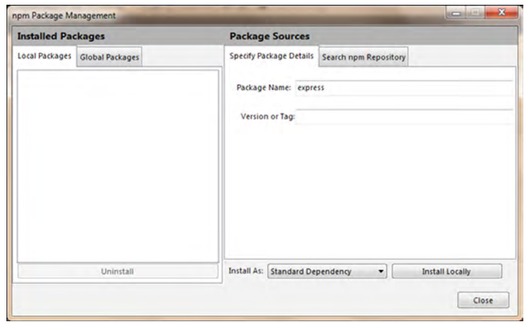


Figure 5: NPM Package Manager

Using the above dialog, we can do the following:

- View all local NPM packages under the 'Local Packages' tab. Any package installed using the dialog when Local Packages is selected, gets installed locally in the project

- View Packages installed globally. These packages are available for all Node.js applications. Any package installed when under 'Global packages' tab, is installed globally

- If you know the name and version of the package to be installed, you can specify the details in them under "Specify Package Details" tab and install it

- You can search for a package and install its latest version using the 'Search npm Repository' tab

- Using the 'Install As' drop down (at the bottom), the package can be installed either as:

  - Standard dependency: Package gets listed in package.json file and will be installed automatically if any of the mentioned packages is missing when the project is opened in Visual studio

  - Dev dependency: Package gets listed under *devDependencies* in package.json. These packages are installed and used only during development

  - Optional dependency: Package is listed in *optionalDependency*. These packages are not mandatory for the application

Let's install express.js as a standard dependency locally. After installing the package, notice the Solution Explorer after installing the package. Express depends on a number of other packages; they are all installed and displayed under the express package.

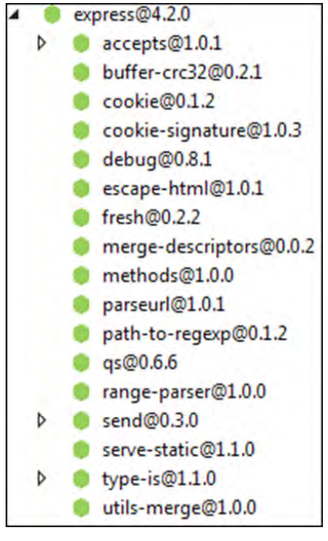


Figure 6: Express.js and dependencies

Along with it, we need to install another NPM package body-parser which is a Node.js body parsing middleware. Once done with this, replace the code in server.js with the following:

```
var http = require('http');
var express = require('express');
var bodyParser = require('body-parser');

var port = process.env.port || 1337;

var app = express();
app.use(bodyParser());

app.get('/', function (request, response) {
    response.send("<b>This response is generated from
Express router!!</b>");
});

app.listen(port);
```

Editorial Note: *Use bodyParser with caution as it contains a vulnerability using which attackers can fill up your disk space by creating an unlimited number of temp files on the server. For a possible solution check http://andrewkelley.me/post/do-not-use-bodyparser-with-express-js.html*

The above code starts the Node.js server at port 1337 and when it gets a request at the home URL, it responds with the HTML we specified in response.send. Instead of creating the server on our own using HTTP, we are now relying on Express.js to do that.

Let's add an HTML page to the site and use it in response to the request to the home URL. Add a new folder to the project and change its name to 'views'. In this folder, add a new HTML file and change its name as 'MoviesList.html' and place the following mark-up in the file:

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <b>Node-Express Movie List</b>
</body>
</html>
```

Now modify the response to home URL as:

```
app.get('/', function (request, response) {
    response.sendfile("views/MoviesList.html");
});
```

Save and run the application. You will see the MoviesList.html getting rendered in your browser.

Let's create a simple REST API that sends some JSON data back to the client. Add the following snippet to server.js:

```
app.get('/api/list', function (request, response) {
    response.send(
        { "movieId": 1, "name": "The Pacific Rim" },
        { "movieId": 2, "name": "Yeh Jawani Hai
Deewani" });
});
```

In your browser, enter the URL http://localhost:1337/api/list. You will see that the JSON data sent above gets displayed on the browser. Similarly, you can create APIs for POST, PUT, DELETE or PATCH by replacing get in the above snippet with the corresponding method.

But working with hard-coded in-memory objects is no fun. Let's use storage instead. With Node.js, it is preferred to use a NoSQL database rather than a traditional RDBMS based database. This is because most of the NoSQL databases work directly with JSON and the framework doesn't have to worry about converting the data. MongoDB is the most widely used database with Node.js. Let's briefly look at MongoDB before we start using it in our application.

INTRODUCTION TO MONGO DB

Mongo DB is a NoSQL database. It stores data in the form of BSON (Binary JSON) documents. Unlike traditional RDBMS databases, NoSQL databases need not know about the structure of the data stored in them. The same holds true for MongoDB as well. MongoDB needs a folder to be mapped with, where it creates files to store the data.

To continue further, install and setup MongoDB (if you haven't already) by following the instructions specified at the beginning of this article.

Now, open a command prompt and move to the folder where the binaries of MongoDB are located and run the command mongo

```
>mongo
```

To check which database we are connected to, just type *db* and hit enter.

```
>db

test
```

So 'test' is the default database used by MongoDB. To create a DB, just type *use* followed by the DB name you would like to create:

```
>use studentDb
```

use does one of the two things:

- If the DB exists, it points to the DB
- If the DB doesn't exist, it creates the DB and then points to it

If you run the *db* command now, you should see studentDb. In MongoDB we don't create tables; rather, we create collections. To see how to create a collection and work with it, let's create a collection to store student information. The commands that we write in MongoDB don't look like SQL commands. They look like function call chains or LINQ queries. The following statement creates a collection and adds data to it:

```
>db.students.insert({"studentId":1,"name":"Ravi"})
```

```
Let's see if the collection is created and if it has
any data in it:

>show collections  : Lists all collections in the
current DB

>db.students.find()  : Shows contents of the collection
students
```

Now insert another row, with a different structure:

```
>db.students.insert({"studId":2,"firstname":"Harini",
"lastname":"Kumari","dob":"05-29-1987",
"occupation":"Govt Employee"})
```

To check data in the collection, we have to execute the find() function on the collection:

```
>db.students.find()
```

As you see, to identify each entry in the collection, MongoDB adds the property _id and assigns a unique value to it. To remove all entries from the collection, execute the following command:

```
>db.students.remove()
```

Now you have some basic idea on how to use MongoDB. This understanding is just enough to continue further. If you want to master MongoDB, check the documentation on their official site.

INTERACTING WITH MONGODB FROM NODE.JS

Like any other DB, we need a driver to interact with MongoDB from Node/Express. The most popular Node.js driver for MongoDB is *Mongoose*. It can be installed from NPM. Let's use the command line to install this package:

```
npm install mongoose --save
```

Check the NPM node in the Solution Explorer. You should be able to see the new package there. NTVS (NodeJS tools for Visual Studio) is smart enough to detect any new NPM package and immediately display it in the IDE.

Create a new folder and change the name of the folder as 'server'. We will place our server code in this folder, to keep the code separated. Add a new JavaScript file to this folder and change the name to 'MongoOperations.js'. To work with Mongoose, just like the other packages, we need to load the module using *require*:

```
var mongoose = require('mongoose');
```

To connect to a database, we need to invoke the connect method with a url:

```
mongoose.connect("mongodb://localhost/moviesDb");
var db=mongoose.connection;
```

If the DB doesn't exist yet, MongoDB will create it for us. Mongoose is almost like an ORM. It needs schema of the data to work with a collection stored in MongoDB. This doesn't sound good, because it induces the strictness that the database itself doesn't have. But, it still is the most popular driver for MongoDB for Node.js.

Let's create a schema to store details of the movies. Each movie would have three fields: name of the movie, a Boolean field showing if the movie is released and a Boolean field showing if you have watched it already.

```
var movieSchema = mongoose.Schema({
    name: String,
    released: Boolean,
    watched: Boolean
});
var MovieModel = mongoose.model('movie', movieSchema);
```

Notice that I didn't add an ID field to the collection; we will use the _id field that is automatically added by MongoDB.

When the application runs for the first time, the collection wouldn't have any value. Let's seed some data into the collection if the collection is empty. We can do it by listening to the open event on the database object created above.

```
db.on('error', console.error.bind(console, "connection
error"));
db.once('open', function () {
    console.log("moviesDb is open...");
    MovieModel.find().exec(function (error, results) {
      if (results.length === 0) {
        MovieModel.create({ name: "The Amazing Spider-
        Man 2", released: true, watched: false });

        MovieModel.create({ name: "The Other Woman",
        released: true, watched: true });

        MovieModel.create({ name: "Shaadi ke Side
        Effects", released: false, watched: false });

        MovieModel.create({ name: "Walk of Shame",
        released: true, watched: false });

        MovieModel.create({ name: "Lucky Kabootar",
        released: false, watched: false });
      }
    });
});
```

Before adding data to the collection, we are checking if the collection already has any data to prevent duplicates. You can replace names of the movies with your favourite ones :)

Also, notice the pattern of execution of the find() method in the above snippet. The method exec() is executed asynchronously. The first parameter that it sends to its callback is *error*, which is set in case the operation fails. The second parameter is the result of the operation. All asynchronous operations in Node.js follow the same pattern.

In the application, we will perform the following operations on movies:

1. Adding a new movie
2. Listing movies
3. Modifying released and watched status of the movie

As a first step, add *require* statement to server.js to include MongoOperations.js:

```
var mongoOps = require('./server/MongoOperations.js');
```

Following are the REST API methods that respond to GET, POST and PUT operations on movies collection:

```
app.get('/api/movies', mongoOps.fetch);

app.post('/api/movies', mongoOps.add);

app.put('/api/movies/:movieId', mongoOps.modify);
```

We are yet to define the fetch, add and modify methods in MongoOperations.js. To get the list of movies in the DB, we need to invoke find() method. Similarly to add a new movie, we need to invoke the create() method on the Mongoose model. The add() method looks like the following:

```
exports.fetch = function (request, response) {
    MovieModel.find().exec(function (err, res) {
        if (err) {
            response.send(500, { error: err });
        }
        else {
            response.send(res);
        }
    });
};

exports.add = function (request, response) {
    var newMovie = { name: request.body.name, released:
    false, watched: false };

    MovieModel.create(newMovie, function (addError,
    addedMovie) {
        if (addError) {
            response.send(500, { error: addError });
        }
        else {
            response.send({ success: true, movie:
            addedMovie });
        }
    });
};
```

Similarly, for modifying we need to call the update() method. But this method needs a condition to be applied to fetch rows to be updated and to check if the update is on multiple rows.

```
exports.modify = function (request, response) {
    var movieId = request.params.movieId;
    MovieModel.update({ _id: movieId }, { released:
    request.body.released, watched: request.body.
    watched }, { multi: false },
        function (error, rowsAffected) {
            if (error) {
                response.send(500, { error: error });
            }
            else if (rowsAffected == 0) {
                response.send(500, { error: "No rows
                affected" });
            }
            else {
                response.send(200);
            }
        }
    );
};
```

With this, we are done with our work on server side. Let's build the client components now.

DESIGN A HTML PAGE FOR MOVIE LIST

I chose Angular to write the client-side JavaScript for several good reasons. Angular JS makes it easy to interact with REST APIs and data binding support that it brings in, is a cool breeze. You can refer to articles in older versions of DNC Magazine to learn more on Angular JS (January 2014 edition for Angular JS with ASP.NET MVC and May 2014 edition for Writing Angular code with TypeScript).

The UI would be a simple HTML page that has the following components:

- a textbox to accept name of new movie that you want to watch
- one column listing released movies
- another column listing upcoming movies

Dividing the movies into two sets is very easy if you use Angular's filters with repeater binding. Following is the HTML mark-up of the page:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Node-Express Movie List</title>
    <link rel="stylesheet" href="//netdna.bootstrapcdn.
com/bootstrap/3.1.1/css/bootstrap.min.css" />
    <link rel="stylesheet" href="/styles/site.css" />
</head>

<body>
  <div class="container">
    <div class="text-center" ng-app="moviesApp"
    ng-controller="MoviesCtrl">
      <h1>Node-Express Movie List</h1>
      <div class="col-md-12 control-group">
        <input type="text" style="width:
        200px;" ng-model="newMovieText" />
        <button id="btnAddTodo" class="btn"
        style="margin: 2px;"
        ng-click="addMovie()"
        ng-disabled="!newMovieText">Add Movie
        </button>
      </div>
    </div>

    <div class="col-md-5 sticky-note">
      <h3 class="text-center">Released
      Movies</h3>
      <div class="col-md-5 rowmargin
      todoItem" ng-repeat="movie in movies |
      filter:{released:true}">
        <div class="thumbnail">
          <input type="checkbox"
          ng-model="movie.watched"
          ng-change="movieWatched(movie)"/>
           
          <span ng-class="{watchedMovie:
          movie.watched}">{{movie.name}}</span>
        </div>
      </div>
    </div>

    <div class="col-md-5 sticky-note">
      <h3 class="text-center">Coming Up...</h3>
      <div class="col-md-5 rowmargin todoItem" ng-
      repeat="movie in movies |
      filter:{released:false}">
        <div class="thumbnail">
          {{movie.name}}
          <br />
          <br />
          <input type="button" value="Released!"
          class="btn btn-link released-button" ng-
          click="movieReleased(movie)" style="" />
        </div>
      </div>
    </div>
  </div>

  <script src="https://ajax.googleapis.com/ajax/libs/
  angularjs/1.2.16/angular.js"></script>
  <script src="/scripts/movies.js"></script>
</body>
</html>
```

Add a folder to the application and change the name to 'public'. This folder would contain all static resource files like JavaScript files, CSS styles for the application. In this folder, add two more folders and name them 'styles' and 'scripts'. We will add scripts

and styles to make our HTML page look similar to the one shown in Figure 7:

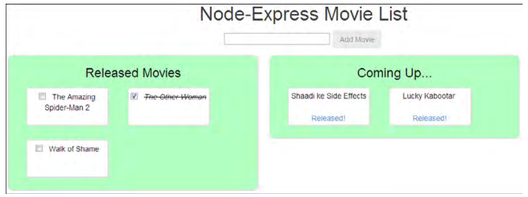


Figure7: Node-Express Movies List

While referring static files in the HTML page, we didn't include the folder path 'public' in it; we need to add the following statement to the server.js to automatically prepend 'public' to the path of the static files:

```
app.use(express.static(path.join(__dirname,
'public')));
```

You can copy the CSS from the downloadable demo app. Let's write the client script to interact with the Node.js API.

ANGULAR JS SCRIPT TO CONSUME REST API AND BIND DATA

Create a module and add a factory to interact with the REST API using $http. The service would contain methods to get, update and add movies.

```
var app = angular.module('moviesApp', []);

app.factory('moviesCRUD', function ($http, $q) {
  function getAllMovies() {
    var deferred = $q.defer();
    $http.get('/api/movies').then(function (result) {
      deferred.resolve(result.data);
    }, function (error) {
      deferred.reject(error);
    });

    return deferred.promise;
  }

  function addMovie(newMovie) {
    var deferred = $q.defer();

    $http.post('/api/movies', newMovie).then(function
(result) {
  deferred.resolve(result.data.movie);
  }, function (error) {
    deferred.reject(error);
  });
  return deferred.promise;
}

function modifyMovie(updatedMovie) {
  var deferred = $q.defer();
  $http.put('/api/movies/' + updatedMovie._id,
  updatedMovie).then(function (data) {
    deferred.resolve(data);
    }, function (error) {
        deferred.reject(error);
    });

    return deferred.promise;
}

return {
    getAllMovies: getAllMovies,
    addMovie: addMovie,
    modifyMovie: modifyMovie
  };
});
```

and finally, we need a controller to respond to the actions on the page. The actions include:

- Getting list of movies at the beginning
- Marking a movie's status as 'released'
- Marking a movie's status as 'watched' or 'not watched'
- Add a new movie

```
app.controller('MoviesCtrl', function ($scope,
moviesCRUD) {
    $scope.released = { released: true };
    $scope.notReleased = { released: false };

    function init() {
        moviesCRUD.getAllMovies().then(function
(movies) {
            $scope.movies = movies;
        }, function (error) {
            console.log(error);
        });
    }

    $scope.movieReleased = function (movie) {

        moviesCRUD.modifyMovie({ _id: movie._id, name:
movie.name, released: true, watched: movie.watched })
                .then(function (result) {
                    if (result.status === 200) {
                        movie.released = true;
                    }
                }, function (error) {
                    console.log(error);
                });
    };

    $scope.movieWatched = function (movie) {
        moviesCRUD.modifyMovie(movie)
                .then(function (result) {
                    if (result.status === 200) {
                        console.log("Movie updated");
                    }
                }, function (error) {
                    movie.watched = !movie.watched;
                });
    };

    $scope.addMovie = function () {
        moviesCRUD.addMovie({ name: $scope.newMovieText
}).then(function (newMovie) {
            $scope.movies.push(newMovie);
            $scope.newMovieText = "";
        }, function (error) {
            console.log(error);
        });
    };

    init();
});
```

And that's our end-to-end application using NodeJS.

DEPLOYING NODEJS APPLICATIONS ON IIS

Now that we have built a small end-to-end application using NodeJS, let's deploy it to a server to keep it running all the time. Internet Information Services (IIS) doesn't support deploying NodeJS applications by default, as IIS doesn't understand NodeJS. To make it understand, we need to install an IIS module. IISNODE is an open-source project that makes it possible to host NodeJS applications on IIS. You can download and install the module from its Github page. Download the installer that suits your system requirements and install the setup. The installer adds a new module to IIS. You should be able to see it in the list of modules in your local IIS:

HttpCacheModule	%windir%\System32\inetsrv\cachhttp.dll
HttpLoggingModule	%windir%\System32\inetsrv\loghttp.dll
HttpRedirectionModule	%windir%\System32\inetsrv\redirect.dll
iisnode	%programfiles%\iisnode\iisnode.dll
IpRestrictionModule	%windir%\System32\inetsrv\iprestr.dll
IsapiFilterModule	%windir%\System32\inetsrv\filter.dll
IsapiModule	%windir%\System32\inetsrv\isapi.dll

Figure 8: IIS Modules list containing iisnode

Process of deploying a NodeJS application in IIS is somewhat similar to that in case of ASP.NET. We need to have a web.config file added to the folder. Add a file to the project and change its name as web.config. First and the most important configuration is to add the server JavaScript file as an HttpHandler.

```
<configuration>
  <system.webServer>
    <handlers>
      <add name="iisnode" path="server.js" verb="*"
      modules="iisnode" />
    </handlers>
  </system.webServer>
</configuration>
```

The web server may receive a request for either static content or a dynamic content. The web.config file should be able to show the right direction to IIS so that the request would be sent to the right destination. The following URL rewriting rules are used for this purpose:

```
<rewrite>
  <rules>
    <rule name="StaticContent">
      <action type="Rewrite" url="public{REQUEST_URI}"/>
    </rule>
    <rule name="DynamicContent">
      <conditions>
        <add input="{REQUEST_FILENAME}"
```

```
        matchType="IsFile" negate="True"/>
      </conditions>
      <action type="Rewrite" url="server.js"/>
    </rule>
  </rules>
</rewrite>
```

But this rule doesn't work for HTTP PUT requests by default. We need to add the following configuration under system.webServer to make it work:

```
<modules>
  <remove name="WebDAVModule"/>
</modules>
```

Now you can deploy the application to IIS like any other ASP.NET application. Open IIS, right click on Default Web Site and choose Add New Application:

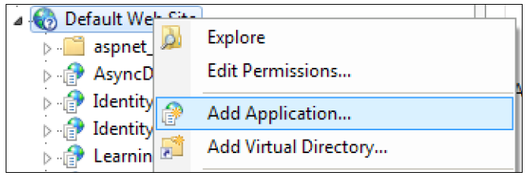


Figure 9: Adding new application in IIS

Enter the name and select the physical location of the NodeJS application to be deployed. Physical path of the folder should be the one that contains the web.config file. Choose an application pool of your choice and click OK.

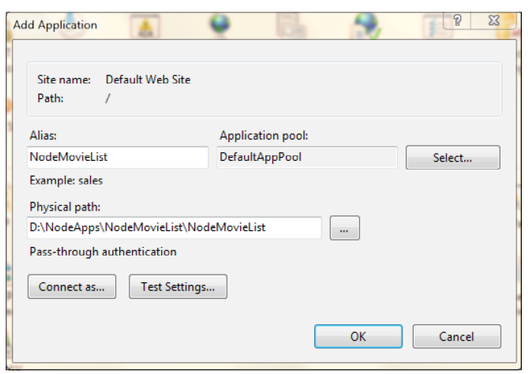


Figure 10: Congifuring path and application pool in IIS

Browse the application in your favorite browser and you should be able to see the same screen as you saw earlier.

CONCLUSION

We saw how Node.js Tools for Visual Studio makes it easy to develop an end-to-end JavaScript application in Visual Studio. As you keep using it, you will find the value that this toolset brings in. You may also face some difficulties as the toolset is still in beta. But Node as a platform has a huge community and the ecosystem is growing by heaps. Though Visual Studio was a .NET IDE till sometime back, Microsoft has been investing a lot of time and resources to make it single IDE for developing almost anything you can think of. NTVS is one such project that demonstrates this effort from Microsoft. IISNODE brings NodeJS closer to .NET developers by making it possible to deploy the NodeJS apps on IIS. IISNODE is used by hosting providers like Azure to support hosting of NodeJS application easier.

Node is open source, it's awesome, and it now comes with .NET. What else do you need to get started? ■

Download the entire source code from our GitHub Repository at bit.ly/dncm13-nodejs

Ravi Kiran is a developer working on Microsoft Technologies. These days, he spends his time on the front-end JavaScript framework Angular JS and server frameworks like ASP.NET Web API and SignalR. He actively writes what he learns on his blog at sravi-kiran.blogspot.com. He is a DZone MVB. You can follow him on twitter at @sravi_kiran

PIC GALLERY DNCMAGAZINE

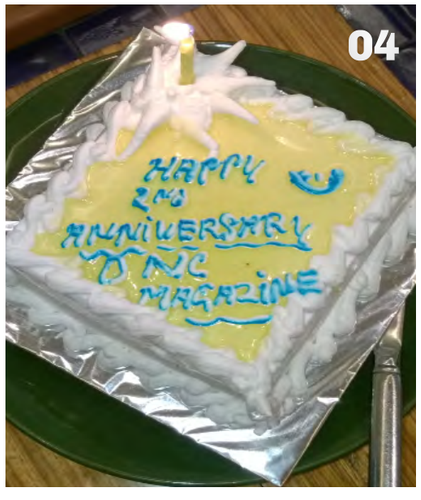
01 Everyone listening to Subodh Sohoni share his ALM implementation stories.

02 Mahesh and Vikram share a light joke. Mahesh is fondly called 'All-in-one' for his Web, Desktop and Phone Dev Skills. This yr its the Cloud!

03 Flashing Smiles, just for you! Happy Birthday #DNCMag

04 Here's a Wish! We want to grow so old celebrating our Anniversary Editions that one day our candles will start costing more than the cake!

05 When is a cake like a golf ball? When it's been sliced :) Gouri and Minal doing the honors.



ANNIVERSARY PARTY 2014

06 Raising a toast to our readers, sponsors and authors (cc: Craig, Filip, Gregor, Mayur, Ravi, Sumit and many more)

07 Give a man a Barbecue, feed him for a day; Teach a man to Barbecue and feed him for the summer!

08 'Real' Chefs celebrating our 'Curry' Chefs! From RtoL: Gouri, Suprotim, Praveen, Vikram, Mahesh and Subodh

09 Wondering why is everybody blushing?

10 "The DNC Magazine Familia", From R to L: Gouri Sohoni, MInal Agarwal, Suprotim, Pravinkumar, Vikram Pendse, Mahesh Sabnis and Subodh Sohoni.

Using MVVM Light in WPF

for Model-View-ViewModel implementation

Separation of Concerns (decoupling) or SoC is a principle that promotes best practices for software architecture. A 'concern' can be considered as an aspect of software functionality. For eg: The UI can be a concern, the business logic can be another concern and so on. The idea is to make sure each concern has a single, well-defined purpose and to find a balanced way to separate these features and concepts, into different modules. This ultimately reduces duplicate code and keeps the modules decoupled from each other, making our application maintainable and manageable. As a very basic example of SoC, think about HTML, CSS and JavaScript, where all these technologies have a well-defined purpose. HTML defines the content structure, CSS defines content presentation and JavaScript defines how the content interacts and behaves with the user.

To realize the SoC principle, many Design Patterns have emerged over the years. For example, Model-View-Presenter (MVP) is suited for Windows Forms; Model-View-Controller (MVC) for ASP.NET MVC; Model-View-ViewModel (MVVM) works well with WPF and so on. For those interested, there's a good article by Martin Fowler which explains the differences in these patterns over here: http://martinfowler.com/eaaDev/uiArchs.html.

WHAT IS MODEL-VIEW-VIEWMODEL (MVVM)?

XAML *enables* SoC in WPF, Silverlight, Windows Phone, and Windows 8 apps by separating the GUI of the application from the programming logic (coded in C #or VB.NET). Model-View-ViewModel (MVVM) is a design pattern that *addresses* SoC by allowing separation of your Business logic from the View (UI), which ultimately makes it easier to write unit tests and enables parallel development and design. It leverages the rich data binding capabilities of the XAML platform to expose the view model to the UI through the view's *DataContext* property. The business layer is also termed as Model whereas the *ViewModel* layer is responsible for exposing data objects from model to the UI using DataBinding. The *ViewModel* contains the View display logic where actions on UI can be handled using Commands properties declared in *ViewModel*.

WHY MVVM LIGHT?

In order to implement MVVM, you need to first understand commanding, messaging and binding. Then you need to understand the MVVM principles and implement these principles keeping a nice balance of power and simplicity in your development. You also need to provide Unit Testing support. All in all, this takes a considerable amount of time and efforts. Luckily there are some nice MVVM frameworks to choose from like Prism, Cailburn, nRoute and Galasoft's MVVM Light Toolkit. We will be exploring how to implement MVVM in WPF applications using the Galasoft's MVVM Light Toolkit by Laurent Bugnion.

The main purpose of the MVVM Light toolkit is to accelerate the creation and development of MVVM applications in WPF, Silverlight, Windows Store (RT) and for Windows Phone

The steps explained in this article are targeted specifically towards those who want to start with MVVM development and require a readymade toolkit for developing their WPF applications.

INSTALLING THE MVVM LIGHT

The MVVM Light toolkit can be downloaded from https://mvvmlight.codeplex.com/.

Project templates for Visual Studio 2012 and 2013 can be downloaded at http://mvvmlight.codeplex.com/releases/view/115541 . Currently the templates are only provided for Visual Studio 2012 and 2013 for the Pro, Premium and Ultimate editions. The MvvmLight.VS2012.vsix is for Visual Studio 2012 and MvvmLight.VS2013.vsix is for Visual Studio 2013. Depending on your versions of Visual Studio, once the respective template is installed, the project template will be available as shown in Figure 1:

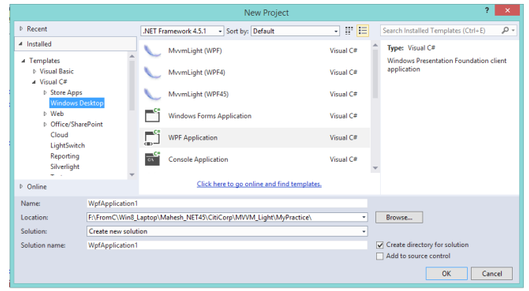


Figure 1: MVVM Light Toolkit template in VS 2013

These project template by default provides the necessary libraries for the MVVM Light Framework with the project structure containing ViewModel classes, Models as shown in Figure 2:

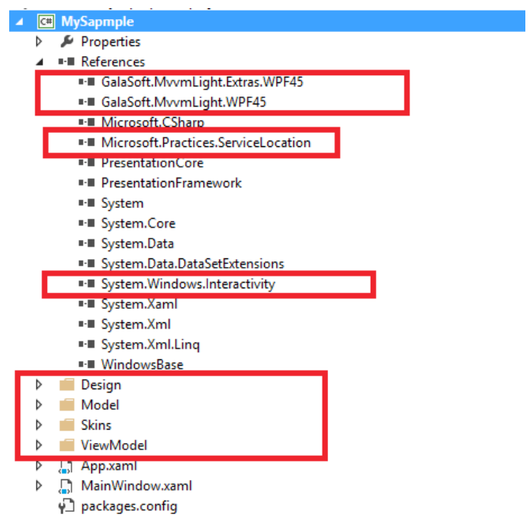


Figure 2: MVVM Light libraries in Visual Studio

The libraries provide classes for implementing ViewModels

with notifiable properties, Command, etc.

If we need to add MVVM Light libraries in an already existing project, then we can make use of the NuGet package to get these libraries. To do so, open an existing WPF project in Visual Studio > right-click on the project > select *Manage NuGet Package* > and select MVVM Light libraries from the NuGet Window as shown in Figure 3:

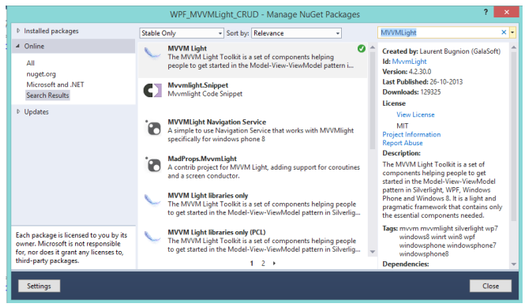


Figure 3: MVVM Light NuGet Package

USING MVVM LIGHT IN WPF 4.5

In the following steps, we will make use of MVVM Light for implementing a WPF application that performs some basic database operations. In these steps, we will make use of the following features of MVVM Light:

- Creating ViewModel using *ViewModelBase* class.
- Defining Notifiable properties in ViewModel class and raising PropertyChanged event on them using *RaisedPropertyChanged* method from the ViewModelBase class.
- Using *RelayCommand* class provided by *Commanding* for method execution based upon actions taken on UI.
- Using *EventToCommand* class provided by *Commanding* to defining commands on the WPF elements that do not by default support the *Command* property.
- Messenger feature for exchanging messages across objects.

For this application, we will be using the following table in a sample SQL Server Company database:

	Column Name	Data Type	Allow Nulls
🔑	EmpNo	int	☐
	EmpName	varchar(50)	☐
	Salary	decimal(18, 0)	☐
	DeptName	varchar(50)	☐
	Designation	varchar(50)	☐
			☐

Figure 4: SQL Server Employee table

Step 1: Open Visual Studio and create a WPF Application and name it 'WPF_MVVMLight_CRUD'. To this project, add the MVVM Light Libraries using NuGet Package as discussed in the installation section. The project will add necessary libraries and the 'ViewModel' folder with the following classes:

- ***MainViewModel.cs*** - This class is inherited from ViewModelBase class and it provides access to *RaisedPropertyChanged* method for notifiable properties.

- ***ViewModelLocator.cs*** - The ViewModelLocator class contains static references for all the view models in the application. The constructor of this class provides a very simple IOC container for registering and resolving instances. Here's a code sample:

```
ServiceLocator.SetLocatorProvider(() => SimpleIoc.
Default);
```

Information about the *ServiceLocator* can be found over here: http://msdn.microsoft.com/en-us/library/ff648968.aspx

The class registers the *MainViewModel* class in the IOC container in its constructor :

```
SimpleIoc.Default.Register<MainViewModel>();
```

It also provides an instance of the MainViewModel using read-only property as seen here:

```
public MainViewModel Main
{
    get
    {
        return ServiceLocator.Current.
        GetInstance<MainViewModel>();
    }
}
```

The *ViewModelLocator* instance will be registered in the App.Xaml resources:

```
<vm:ViewModelLocator x:Key="Locator"
                     d:IsDataSource="True" />
```

This will be used for DataBinding across views in the application.

Step 2: In the project, add a new folder with the name 'Model'. In this folder, add a new ADO.NET Entity Data Model with the name 'CompanyEDMX' > select the SQL Server Database and the EmployeeInfo table. After completion of the wizard, the following table mapping will be displayed:

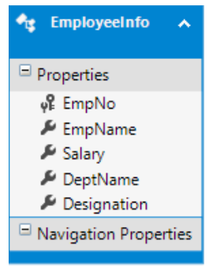


Figure 5: Employee Table Mapping

Step 3: Now add a new folder called 'Services' and in this folder, add a class file with the following code:

```
using System.Collections.ObjectModel;
using WPF_MVVMLight_CRUD.Model;

namespace WPF_MVVMLight_CRUD.Services
{

    /// <summary>
    /// The Interface defining methods for Create
        Employee and Read All Employees
    /// </summary>
    public interface IDataAccessService
    {
        ObservableCollection<EmployeeInfo>
        GetEmployees();
        int CreateEmployee(EmployeeInfo Emp);
    }

    /// <summary>
    /// Class implementing IDataAccessService interface
    /// and implementing its methods by making call to
    /// the Entities using CompanyEntities object
    /// </summary>
    public class DataAccessService : IDataAccessService
    {
        CompanyEntities context;
        public DataAccessService()
        {
            context = new CompanyEntities();
        }
        public ObservableCollection<EmployeeInfo>
        GetEmployees()
        {
            ObservableCollection<EmployeeInfo>Employees
            = new ObservableCollection<EmployeeInfo>();
            foreach (var item in context.
            EmployeeInfoes)
            {
                Employees.Add(item);
            }
            return Employees;
        }
        public int CreateEmployee(EmployeeInfo Emp)
        {
            context.EmployeeInfoes.Add(Emp);
            context.SaveChanges();
            return Emp.EmpNo;
        }

    }
}
```

This code defines an interface for accessing data from the database using Entity Framework.

Step 4: To register the Data Access service in the IoC, we need to register the *DataAccessService* class in it. To do so, open the ViewModelLocator class, and add the following line:

```
SimpleIoc.Default.Register<IDataAccessService,
DataAccessService>();
```

The namespace for the DataAccessService class must be used in the ViewModelLocator class.

Step 5: Let's implement the logic for reading all Employees from the table.

In the MainViewModel class, add the following Public notifiable property:

```
ObservableCollection<EmployeeInfo> _Employees;

    public ObservableCollection<EmployeeInfo> Employees
    {
        get { return _Employees; }
        set
        {
            _Employees = value;
            RaisePropertyChanged("Employees");
        }
    }
```

This property will be exposed to the UI. The *setter* of the property calls the *RaisedPropertyChanged* method which will internally raise the *PropertyChanged* event when the data from the collection changes.

Define the *IDataAccessService* object at the ViewModel class level as shown here:

```
IDataAccessService _serviceProxy;
```

In the ViewModel class, declare the following method to fetch Employees data:

```
    /// <summary>
    /// Method to Read All Employees
    /// </summary>
    void GetEmployees()
    {
        Employees.Clear();
        foreach (var item in _serviceProxy.
        GetEmployees())
        {
            Employees.Add(item);
        }
    }
```

The above method calls *GetEmployees()* method from the *DataAccessService* class and puts all Employees in the Employees observable collection.

In the *ViewModel* class, now define *RelayCommand* object as shown here:

```
public RelayCommand ReadAllCommand { get; set; }
```

Use the constructor dependency for passing the *IDataAccessService* to the ViewModel's constructor. The object of the DataAccessService will be available from the IoC which we have registered in Step 4. Also instantiate the *Employees* observable collection and *ReadAllCommand* object:

```
public MainViewModel(IDataAccessService servPxy)
{
    _serviceProxy = servPxy;
    Employees = newObservableCollection<EmployeeInfo>();
    ReadAllCommand = new RelayCommand(GetEmployees);
}
```

The *ReadAllCommand* is passed with the *GetEmployees()* method.

Step 6: In the project, add a new MVVM View provided by the MVVM Light toolkit as seen here:

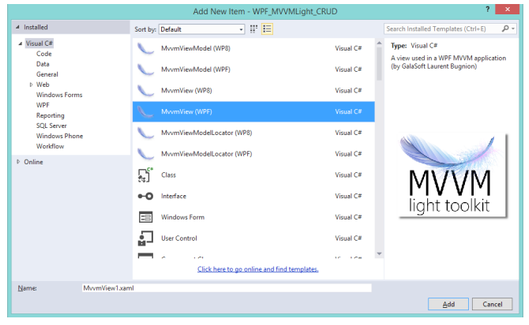


Figure 6: MvvmView template

Name this view 'EmployeeInfoView.xaml'.

Note: *By default, the MVVM Light View adds a WPF Window, so for this application we will be replacing the Window with a UserControl. Once you change the root tag of the View from Window to UserControl, in the code behind of the view, change the base class from Window to UserControl.*

Step 7: In this view, add a DataGrid, TextBlock and a Button as shown in Figure :

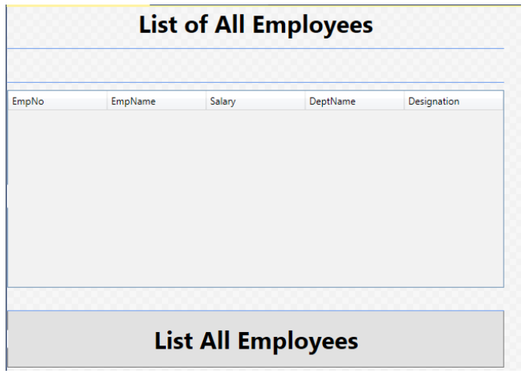


Figure 7: UI Design of our application

In the XAML part, set the *DataContext* property of the UserControl to the 'Main' property exposed by the *ViewModelLocator* class:

```
DataContext="{Binding Main, Source={StaticResource
Locator}}"
```

The 'Locator' is declared in the App.Xaml resources. *Main* is the public property exposed by the ViewModelLocator class which returns an object of the MainViewModel class. The above expression means the MainViewModel is now bound with the UserControl. This means that all public declarations (Notifiable properties and Commands) can be bound with the XAML elements on the View.

Bind the *ReadAllCommand* command and the Employees collection properties of MainViewModel to the Button and DataGrid respectively :

```
<Button x:Name="btnloadallemployees" Content="List
All Employees" Grid.Row="3" FontSize="30"
FontWeight="Bold" Command="{Binding ReadAllCommand}"/>
    <DataGrid x:Name="dgemp" Grid.Row="2"
    ItemsSource="{Binding Employees}"
    ColumnWidth="*"  Margin="0,10,0,28"
    RenderTransformOrigin="0.5,0.5" IsReadOnly="True" >
    </DataGrid>
```

Step 8: Open MainWindow.xaml and change its width to 1300. Set the width of the Grid element to 1290 and create two columns in it, each of width 640. In the zeroth (0) column, add the EmployeesInfo view.

To add the View in the MainWindow.xaml, the namespace of the View must be registered in the Window tag:

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

Now add the View to the Grid:

```
<Views:EmployeeInfoView  Grid.Column="0"/>
```

Step 9: Run the project and a View gets displayed. Click on 'List All Employees' and the result is as shown in the Figure 8:

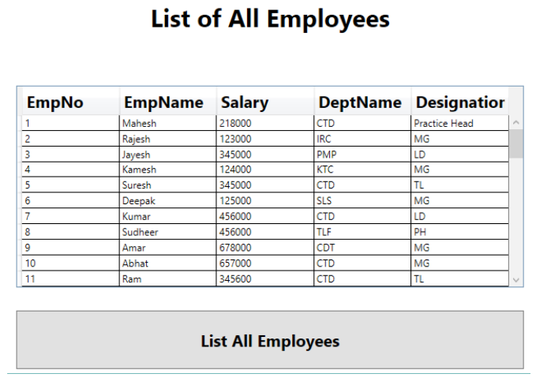


Figure 8: List all employees

PASSING PARAMETERS FROM VIEW TO THE VIEWMODEL

In the previous steps, we discussed how to create ViewModel, defining notifiable properties and the RelayCommand. In this section, we will discuss how to send data from View to the View Model and write in into our database table.

Step 1: In the MainViewModel, declare the following property:
EmployeeInfo _EmpInfo;

```
public EmployeeInfo EmpInfo
{
    get { return _EmpInfo; }
    set
    {
        _EmpInfo = value;
        RaisePropertyChanged("EmpInfo");
    }
}
```

The *EmployeeInfo* object will be used to add new Employee records.

Define the following method which accepts the EmployeeInfo object and saves it to the table by calling the *CreateEmployee()* method from the DataAccessService class.

```
void SaveEmployee(EmployeeInfo emp)
{
    EmpInfo.EmpNo =  _serviceProxy.CreateEmployee(emp);
    if(EmpInfo.EmpNo!=0)
    {
        Employees.Add(EmpInfo);
        RaisePropertyChanged("EmpInfo");
    }
}
```

The CreateEmployee method returns the EmpNo. If this is not zero, then the *emp* object will be added into the Employees observable collection.

Now define the RelayCommand object property in the ViewModel class:

```
public RelayCommand<EmployeeInfo> SaveCommand { get;
set; }
```

The *RelayCommand<T>* generic type property is declared where 'T' represent the input parameter; in our case 'T' is of the type EmployeeInfo.

Instantiate EmpInfo and RelayCommand in the ViewModel constructor:

```
EmpInfo = new EmployeeInfo();
SaveCommand = new
RelayCommand<EmployeeInfo>(SaveEmployee);
```

The RelayCommand is passed with the *SaveEmployee()* method. This is possible because the SaveEmployee method accepts EmployeeInfo as its input parameter. This is the same object defined in the declaration of the generic RelayCommand property.

Step 2: Open App.Xaml and in *resources*, add styles for TextBlock and TextBoxes :

```
<Style TargetType="{x:Type TextBlock}">
    <Setter Property="FontSize" Value="20"></Setter>
    <Setter Property="FontWeight" Value="Bold">
    </Setter>
</Style>

<Style TargetType="{x:Type TextBox}">
    <Setter Property="FontSize" Value="20"></Setter>
    <Setter Property="FontWeight" Value="Bold">
    </Setter>
</Style>
```

Since these styles are defined in App.Xaml without any key, they will be applied for all TextBlocks and TextBoxes in the application.

Step 3: In the Views folder of the project, add a new UserControl (you can use MVVM Light View also), and name it as 'SaveEmployeeView.xaml'. In the View, add TextBlocks, TextBoxes and a Button. Set the DataContext of the UserControl to the Main property of the MainViewModel.

```
DataContext="{Binding Main, Source={StaticResource
Locator}}"
```

For all the TextBoxes, bind their Text property to the *EmpInfo* property and bind the *command* property of Button to the *SaveCommand* exposed by the *Main* ViewModel. Along with it, bind the EmpInfo property to the *CommandParameter* property of the button. This is the actual parameter which will be passed from View to ViewModel. The XAML is as shown here:

```
<TextBox Grid.Column="1" TextWrapping="Wrap"
Text="{Binding EmpInfo.EmpNo,Mode=TwoWay}"/>

<TextBox Grid.Column="1" Grid.Row="1"
TextWrapping="Wrap" Text="{Binding EmpInfo.
EmpName,Mode=TwoWay}"/>

<TextBox Grid.Column="1" Grid.Row="2"
TextWrapping="Wrap" Text="{Binding EmpInfo.
Salary,Mode=TwoWay}"/>

<TextBox Grid.Column="1" Grid.Row="3"
TextWrapping="Wrap" Text="{Binding EmpInfo.
DeptName,Mode=TwoWay}"/>

<TextBox Grid.Column="1" Grid.Row="4"
TextWrapping="Wrap" Text="{Binding EmpInfo.
Designation,Mode=TwoWay}"/>

<Button Content="Save Employee" FontSize="20"
FontWeight="Bold" Grid.Row="5" Grid.ColumnSpan="2"
Command="{Binding SaveCommand}"
CommandParameter="{Binding EmpInfo}"/>
```

The View now looks similar to what is shown in Figure 9:

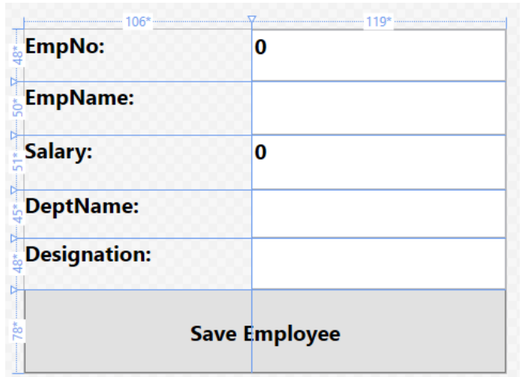


Figure 9: Save Employee View

Step 4: Add the UserControl we just created in MainWindow.xaml in the second column (column index 1):

```
<Views:SaveEmployeeView  Grid.Column="1"/>
```

Step 5: Run the Application, click on the 'List All Employees' to display all employees in the DataGrid.

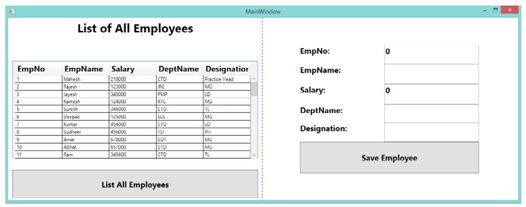


Figure 10: List All Employees View

I have not added any validations to keep the scope of this article concise, but you should. Enter Employee Data (except EmpNo) and click on the 'Save Employee' button. The record gets added. Now scroll to the bottom of the DataGrid and you will find our newly added record:

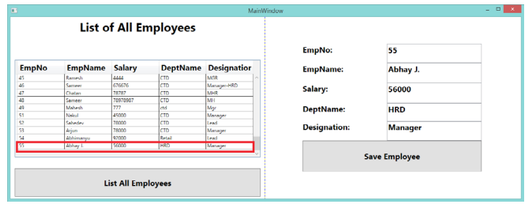


Figure 11: Newly added record

So now we have seen how to make use of generic command to pass parameters from UI to ViewModel.

DEFINING COMMANDING ON UI ELEMENTS 'NOT' HAVING THE COMMAND PROPERTY

UI elements like Button, RadioButton, etc. expose the *command* property, using which methods from the ViewModel can be executed. But what if we need to have a search textbox on the UI in which when the data is entered, matching data from the textbox is searched from a collection, and displayed on the UI? In this case, we need to change the behavior of the textbox to support commanding.

When we add the MVVM Light libraries in the project, the project also adds the *System.Windows.Interactivity.dll* assembly. This assembly allows us to define behaviors for the UI elements.

The MVVM Light library provides an *EventToCommand* class under the *GalaSoft.MvvmLight.Command* namespace. This class allows us to bind any event of any FrameworkElement to ICommand. In our case, we will use the *EventToCommand* to execute a method on the ViewModel class by defining command on the TextChanged event of the TextBox.

Step 1: Open the MainViewModel and add the following property, method and command in it:

```
public string EmpName
{
    get { return _EmpName; }
    set
    {
        _EmpName = value;
        RaisePropertyChanged("EmpName");
    }
}
```

The string property will be bound with the TextBox in the View. This property will be set when the text is entered in the TextBox.

```
void SearchEmployee()
{
    Employees.Clear();
    var Res = from e in _serviceProxy.GetEmployees()
    where e.EmpName.StartsWith(EmpName)
    select e;
    foreach (var item in Res)
    {
        Employees.Add(item);
    }
}
```

The method filters Employees from the collection based upon the EmpName. Now define the RelayCommand object in ViewModel:

```
public RelayCommand SearchCommand { get; set; }
```

Instantiate Command object in the constructor of the MainViewModel by passing the SearchEmployee method to it:

Step 2: Open EmployeeInfoView.xaml and add a TextBlock and TextBox to it. To register the Interactivity and EventToObject to XAML, we need the following assemblies in the UserControl tag:

```
xmlns:i=http://schemas.microsoft.com/expression/2010/
interactivity
xmlns:mvvm=”http://www.galasoft.ch/mvvmlight”
```

Define the DataBinding interactivity for commanding on the TextBox as shown here:

```
<TextBlock HorizontalAlignment=”Left” Margin=”10,7,0,0”
Grid.Row=”1” TextWrapping=”Wrap” Text=”EmpName to
Search:” VerticalAlignment=”Top” Width=”231”/>

    <TextBox HorizontalAlignment=”Left”
     Height=”30” Margin=”262,7,0,0” Grid.Row=”1”
     TextWrapping=”Wrap”  Text=”{Binding EmpName,
     UpdateSourceTrigger=PropertyChanged}”
     VerticalAlignment=”Top” Width=”310”>
        <i:Interaction.Triggers>
            <i:EventTrigger EventName=”TextChanged”>
                <mvvm:EventToCommand
                Command=”{Binding SearchCommand,
                Mode=OneWay}”/>
            </i:EventTrigger>
        </i:Interaction.Triggers>
    </TextBox>
```

In the XAML, the TextBox is bound with the EmpName property of the MainViewModel. The *UpdateSourceTrigger* property of the Binding class is set to the PropertyChanged event; this means that when the text is entered in the textbox, the EmpName property will be set with the text in the textbox. The event trigger is defined inside the textbox for the *TextChanged* event. This means when the TextChanged event is fired, the EventToCommand will execute the SearchCommand method defined in the MainViewModel.

Step 3: Run the application, click on the ‘List All Employees’ button and all employees will be displayed in the DataGrid:

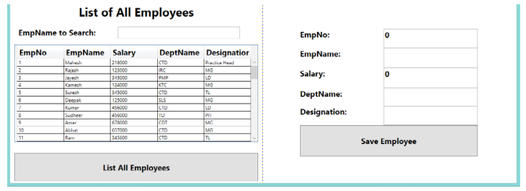


Figure 12: List All Employees

Enter some text in the EmpName to search the TextBox. The DataGrid will be populated by all Employee records having EmpName starting with the text entered in the textbox:

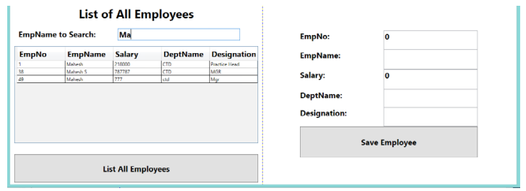


Figure 13: Search Employees and Populate Grid

So using *EventToCommand*, we can easily bind the command to the FrameworkElement.

MANAGING MESSAGING ACROSS TWO VIEWS

Typically when there are multiple Teams involved in the development stage, then there is a possibility that they design separate views and these views are present on the same container. For e.g. take the *EmployeeInfoView* which shows list of all employees and *SaveEmployeeView* which performs Create, Update like operations. Now the requirement is that when an Employee is selected from the EmployeeInfoView, it should be displayed in the SaveEmployeeView for Updation purpose. There is a possibility that since both these views are having their own ViewModels, data will be send across these ViewModels.

Since both views are separate, how do we send the selected employee from one view to other? Traditionally this can be made possible using Event handling mechanism. When an Employee is selected, raise the event and pass the employee information to this event and then listen to this event in some other view. This approach requires that both views should *directly* communicate with each other by having access to each other’s objects. This is a scenario of *tight-coupling*. So now the question is how to implement the same in a *loosely-coupled* manner?

The answer is to use the **MVVM Light Messenger**. This messenger provides a loosely-bound way to send message (data) from one ViewModel to other.

Diagrammatically the Messenger can be explained as:

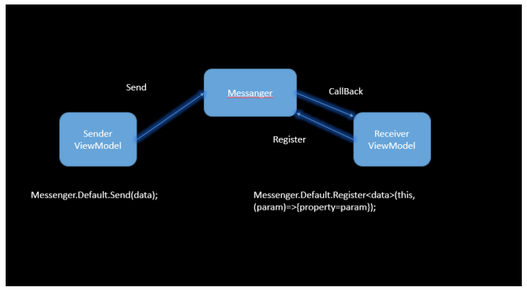


Figure 14: MVVM Light Messenger

Messenger is a *singleton* object which lives throughout the application. The sender ViewModel simply calls the static ‘Send’ method. The receiver ViewModel needs to register with the messenger to receive the object. It provides a call back function which gets called when the new message received. Let’s see how this is done.

Step 1: In the project, add a new folder and name it ‘MessageInfrastructure’. In this folder, add a new class file:

```
using WPF_MVVMLight_CRUD.Model;

namespace WPF_MVVMLight_CRUD.MessageInfrastructure
{
    public class MessageCommunicator
    {
        public EmployeeInfo Emp { get; set; }
    }
}
```

The above class defines *Emp* property of the type EmployeeInfo. This will be used as a message (data) passed from one view to other.

Step 2: In the MainViewModel, add the following method:

```
void SendEmployeeInfo(EmployeeInfo emp)
{
    if(emp!=null)
    {
        Messenger.Default.
        Send<MessageCommunicator>(new
        MessageCommunicator() {
            Emp = emp
        });
    }
}
```

Note: *Please add the ‘GalaSoft.MvvmLight.Messaging’ namespace in the MainViewModel.*

The above method accepts an EmployeeInfo object and calls the Send() method of the Messenger, which is *typed* to the MessageCommunicator class. This means that the View which calls the above method, must pass the EmployeeInfo object.

In the ViewModel define the following *RelayCommand* object:

```
public RelayCommand<EmployeeInfo> SendEmployeeCommand {
get; set; }
```

The RelayCommand is defined with the parameter of the type EmployeeInfo. This means that it will execute method having input parameter of the type EmployeeInfo.

In the constructor of the ViewModel, define an instance of the RelayCommand as shown here:

```
SendEmployeeCommand = new
RelayCommand<EmployeeInfo>(SendEmployeeInfo);
```

Step 2: Open the EmployeeInfoView and define the *EventToCommand* for the DataGrid. Since the DataGrid is bound with the EmployeeInfo collection, when the DataGridRow is selected, it will select the EmployeeInfo object. We will map the *SelectionChanged* event of the DataGrid to the EventToCommand as shown here:

```
<DataGrid x:Name=”dgemp” Grid.Row=”2”
ItemsSource=”{Binding Employees}”
ColumnWidth=”*”  Margin=”0,10,0,28”
RenderTransformOrigin=”0.5,0.5”
IsReadOnly=”True” >
    <i:Interaction.Triggers>
        <i:EventTrigger EventName=”SelectionChanged”>
            <mvvm:EventToCommand
            Command=”{Binding  SendEmployeeCommand,
            Mode=OneWay}”
            CommandParameter=”{Binding
            ElementName=dgemp,Path=SelectedItem}”
             />
        </i:EventTrigger>
    </i:Interaction.Triggers>
</DataGrid>
```

This XAML shows that the Command property is bound with the *SendEmployeeCommand* declared in the ViewModel. The parameter sent from the UI to ViewModel is the SelectedItem, which is an EmployeeInfo object. Since the SendEmployeeCommand executes the SendEmployeeInfo method, the EmployeeInfo object will be passed to this method.

Step 3: Now we need to register for the messenger. To do so, in the MainViewModel add the following method:

```
void ReceiveEmployeeInfo()
{
    if (EmpInfo != null)
    {
        Messenger.Default.
        Register<MessageCommunicator>(this,(emp)=>{
            this.EmpInfo = emp.Emp;
```

The above method registers to the messenger and accepts the Emp message received. This message is then set to the *EmpInfo* notifiable property defined in the ViewModel class. Call this method in the constructor of the ViewModel. Since EmpInfo property is bound with the SaveEmployeeView, the Employee data will be displayed in it.

Step 4: Run the application, click on the Load All Employees button and the DataGrid will show all Employees. Select the Row from the DataGrid, the selected Employee Information will be displayed in the SaveEmployeeView as below:

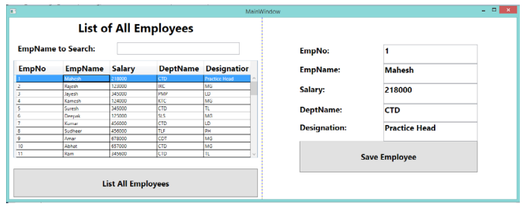


Figure 15: Load All Employees and View Details of each

This shows how easily we can establish Messenger base communication across two separate views.

CONCLUSION

MVVM has lot of advantages, but it can involve a lot of efforts to set things up on your own. The MVVM Light toolkit provides a lot of boiler plate code to implement MVVM based applications quickly and gives the user the freedom to customize and design the application. With the ViewModelBase from the MVVM Light toolkit, we no longer have to implement INotifyPropertyChanged. Additionally the MVVM light toolkit provides Visual Studio templates, a messenger, an IoC container and some useful code snippets that can make our WPF applications shine! ■

Download the entire source code from our GitHub Repository at bit.ly/dncm13-wpfmvvmlight

Mahesh Sabnis is a Microsoft Certified Trainer (MCT) since 2005 and has conducted various Corporate Training programs for .NET Technologies (all versions). Follow him on twitter @maheshdotnet. Mahesh blogs regularly on Azure, SharePoint, Metro UI, MVC and other .NET Technologies at bit.ly/HsS2on

SHAREPOINT BUSINESS CONNECTIVITY SERVICES 2013

In an organization, data is often spread across disparate systems. For example, your data could be residing in SharePoint or could exist in a LOB application like CRM or ERP or even in a Data Warehouse. Accessing this data across different technologies, user interfaces and different systems, in the format you need it to be, can be a daunting task. The solution to this challenge lies in using SharePoint and Business Connectivity Services (BCS).

Business Connectivity Services (BCS) in SharePoint are a set of services and features that enables you to connect SharePoint solutions (via Web Parts, User Profiles or SharePoint lists) to sources of external data. Business Connectivity Services was introduced in SharePoint Server 2010 and has evolved from SharePoint 2007's Business Data Catalog (BDC).

An architectural overview of BCS is shown in Figure 1:

Exploring BCS Connectors and performing CRUD operations using External Content Type

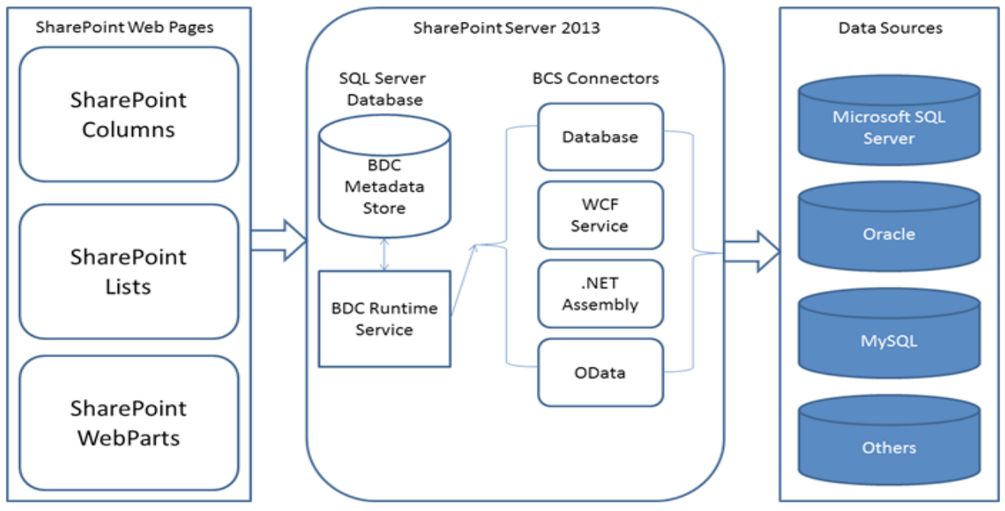


Figure 1 - Architectural Overview of Business Connectivity Services

BCS allows the access of data spread across organization using Web Services, SQL or using OData sources. You can also build custom connectors using .NET Assemblies to access data from different data sources and perform CRUD operations against the data.

There are different ways of presenting external data in SharePoint. The most common way is using an *External List*. Think of External List as something similar to SharePoint list with a difference that it can only be used for displaying external data. You can use an External data column in a SharePoint list and library.

External data can be integrated into SharePoint and Office clients. You can also take this data offline and work with the same. There are various Office client applications which can be used to display or modify the data. Some of them are: Microsoft Word, , Microsoft Access, Microsoft Visio, Microsoft InfoPath, Microsoft Excel and Microsoft Outlook.

BCS CONNECTORS

We have already discussed that BCS enables SharePoint to access data from various external data systems. To access this data into SharePoint, BCS provides a number of connectors.

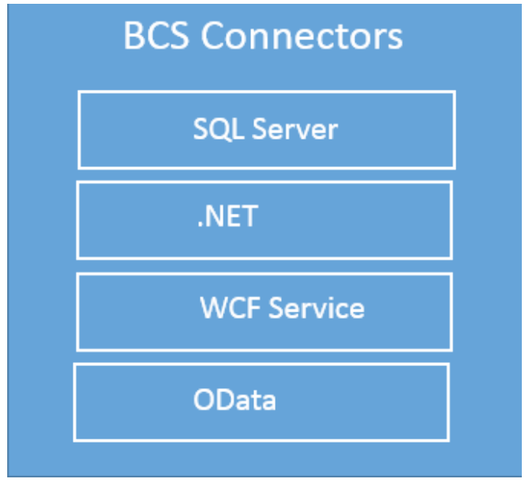


Figure 2 - Different types of BCS Connectors

Let's take a look at each connector:

Database Connector – Using SharePoint Designer 2013, you can connect to a SQL Server database, Oracle or any other databases which supports the OLEDB provider. You can create an External Content Type using SharePoint Designer 2013 as shown in Figure 3:

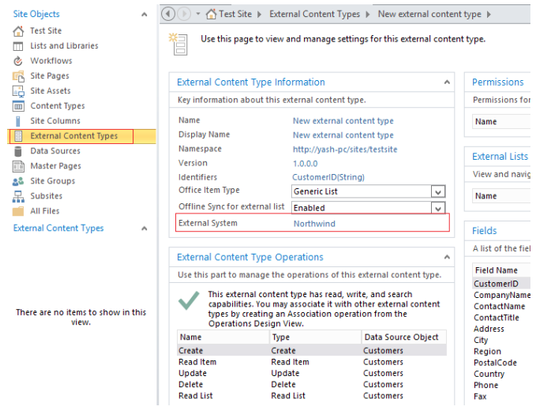


Figure 3 - External Content Type Information

.NET Assembly Connector – Using Visual Studio 2010/2012, you can implement and deploy a .NET Assembly connector which can be connected to any database using custom code. You can also embed some custom logic while designing the .NET connector.

To design the .NET Assembly connector, Visual Studio provides an out-of-box template as shown in Figure 4:

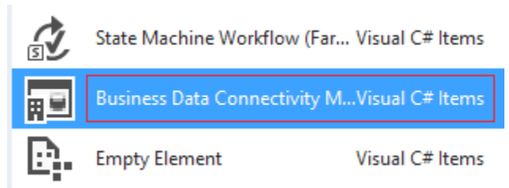


Figure 4 - VS Assembly Connector Template

WCF Service Connector – You can use WCF service connector to connect to various databases. WCF services implements *contracts* which we can use to implement operations performed by using BCS.

Using SharePoint Designer as a tool, you can even discover WCF services. You can configure WCF contracts and binding as shown in Figure 5:

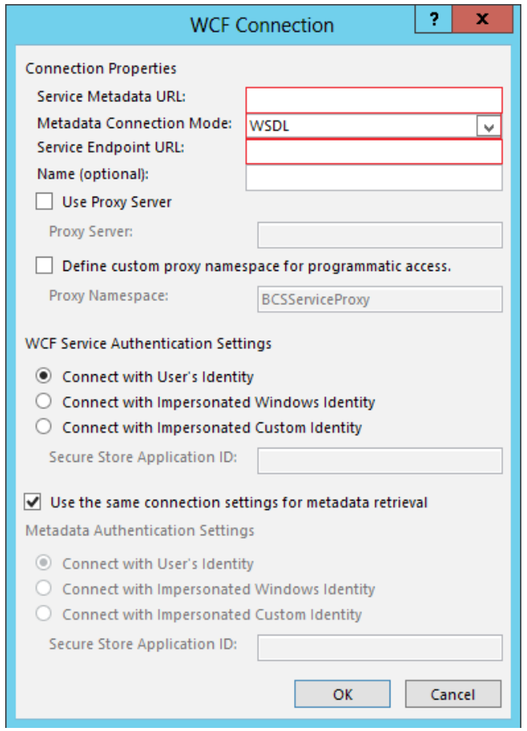


Figure 5 - Configure WCF Services

OData Connector – OData (Open Data Protocol) is a web protocol for performing CRUD operations which is built upon web technologies like HTTP, Atom Publishing Protocol (AtomPub) and JSON to provide access to the data across various applications, services and stores.

In SharePoint 2013, Business Connectivity Services can communicate with OData sources. SharePoint list data is exposed as an OData source, hence this way SharePoint becomes a producer of OData Source.

In this article, we will explore these four connectors and will perform CRUD operations using External Content Type and External List using Visual Studio and SharePoint Designer.

External Content Type (ECT)

External Content Type (ECT) is a core concept of Business Connectivity Services. ECT enables reusability of business entity metadata and provides built-in behaviour for Microsoft Office Outlook items such as Contacts, Tasks, Calendars, Microsoft Word Documents and SharePoint lists and web parts.

By using ECT, information workers need not worry about how complex the External Systems are or how to connect with these systems and location of the systems.

ECT follows the security of the external system as well as SharePoint technology. We can control the data access by configuring security in SharePoint.

We can use External Content Type in SharePoint as a source of the external data in the following way:

1. External Lists in SharePoint which uses ECT as a data source.

2. External Data Column uses ECT as data source to add data to Standard SharePoint Lists.

3. SharePoint Business Connectivity Services offers various web parts like -

   (a) External Data List web part

   (b) External Data Item web part

   (c) External Data Related List web part

CREATING SQL SERVER CONNECTOR USING MICROSOFT SHAREPOINT DESIGNER 2013

We will now see how to fetch data using SQL database connector. For this demonstration, we will make use of SharePoint Designer 2013. I have already created a SharePoint Site using Team Site template.

Open SharePoint Designer and open the SharePoint Site which you have created. Once the site is opened, we will create a new *External Content Type* which is located on the left hand side Navigation pane. The same is shown in Figure 6:

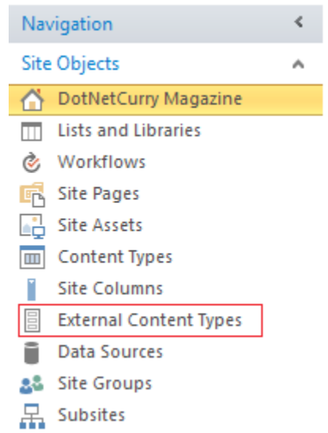


Figure 6 – SharePoint Designer Navigation Pane

Click on External Content Types and make a choice of *External Content Type* from the External Content Types ribbon, as shown in Figure 7:

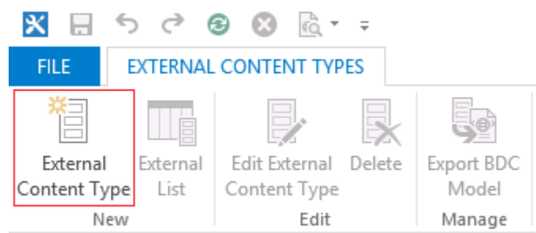


Figure 7 – External Content Type ribbon

Now let's configure ECT as shown in Figure 8:

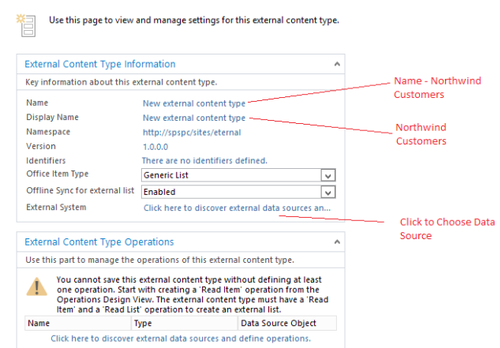


Figure 8 – ETC Configuration

When you click on External Data Sources link to choose the data source, you will see an External Data Source Type Selection dialog box. Choose SQL Server from the dropdown list as shown in Figure 9:

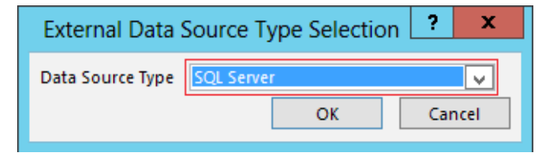


Figure 9 - Choose External Data Source

Now set the SQL Server Connection properties as shown in Figure 10:

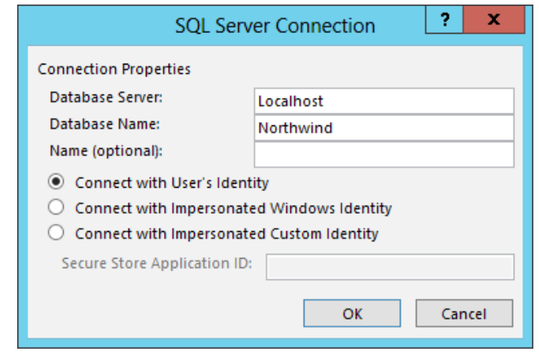


Figure 10 – SQL Server Configuration

As soon as your connection is successful, you will see all the tables, views and stored procedures listed in the Data Source Explorer, as shown in Figure 11:

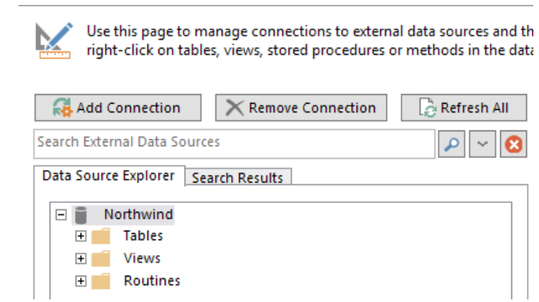


Figure 11 - Northwind Objects

Expand the *Tables* folder and right click the *Customers* table. In the context menu, choose *Create All Operations*. The final view is

shown in Figure 12:

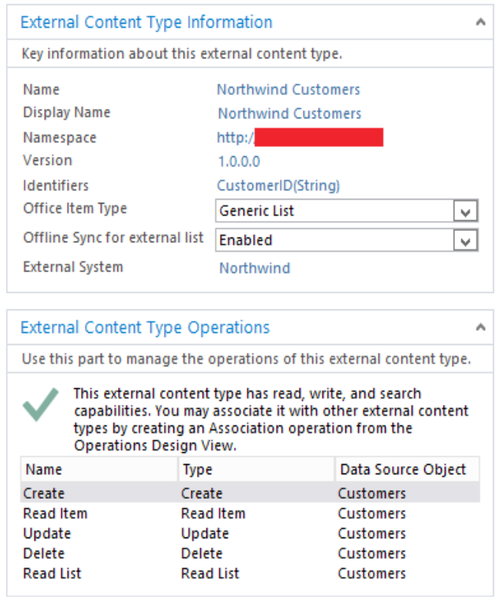


Figure 12 – External Content Type Final View

Now we are ready with all our CRUD operations. We will use this External Content Type in our SharePoint Site. Let's go back to our site and click on *Site Contents* and then click on "Add an App". Choose *External Site* and configure it by giving it a name "SQL Customers" and then choose the *Northwind Customers Content type* as shown in Figure 13:

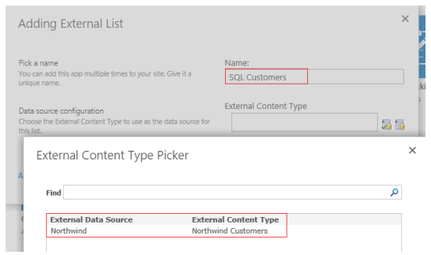


Figure 13 – Create External List

Once you pick up the External Content Type, you will see the following data as shown in Figure 14:

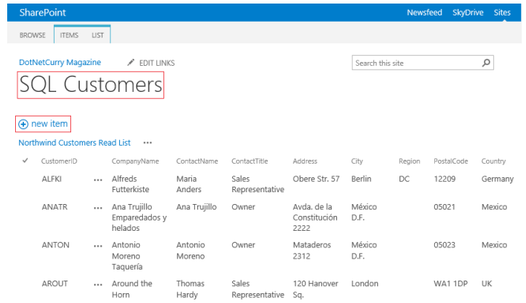


Figure 14 – SQL Customers

CREATING A BCS .NET CONNECTOR USING VISUAL STUDIO 2012/2013

To create a .NET Connector, open Visual Studio and create a SharePoint 2013 Empty Project. I am using Visual Studio 2013 for this demonstration.

Right click the SharePoint Project and add a new item "Business Connectivity Model" as shown in Figure 15:

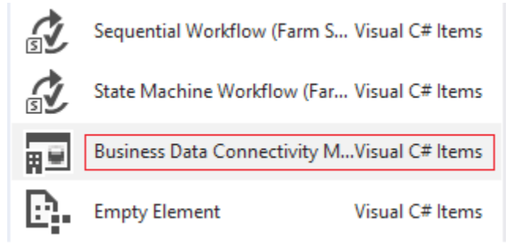


Figure 15 – Create BDC Model

Once the project is ready, we will first prepare our database. I have already created a database with the name "DotNetCurryDatabase". We will now create a table called 'Authors' under the same database. The script for creating the table is as shown here –

```
USE DotNetCurryDatabase
GO
CREATE TABLE Authors
(
    AuthorID INT PRIMARY KEY,
    AuthorName NVARCHAR(100),
    ContactNo NVARCHAR(15),
    City NVARCHAR(30),
    Country NVARCHAR(30)
)

INSERT INTO Authors VALUES(1,'Suprotim Agarwal','+91
6637773677','Pune','India')

INSERT INTO Authors VALUES(2,'Sumit M.','+91
2552622666','Pune','India')

INSERT INTO Authors VALUES(3,'Subodh S.','+91
1222313333','Pune','India')

INSERT INTO Authors VALUES(4,'Gauri S.','+91
4555355555','Pune','India')

INSERT INTO Authors VALUES(5,'Mahesh S.','+91
9999899998','Pune','India')

SELECT * FROM Authors
```

Run this script and test it with the SELECT statement. Now return to Visual Studio where we have added a Business Data Connectivity Model. Rename Entity1 to Author and Identifier1 to AuthorID by changing its data type to Int32 from the properties window as shown in Figure 16:

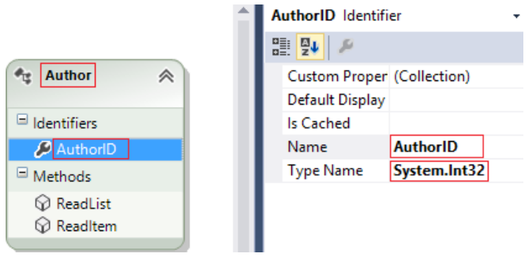


Figure 16 – Modify Entity

Now add a LINQ To SQL class with the name "DotNetCurry" into our project and drag and drop Authors table from Server Explorer from our database DotNetCurryDatabase as shown in Figure 17:

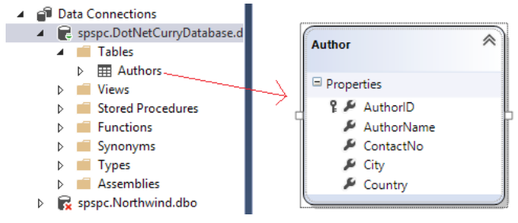


Figure 17 – Authors Table

We will be using LINQ to SQL to perform CRUD operations against our Authors table using ECT. Open BDC Explorer and expand the ReadList method. Rename the Entity1List to AuthorsList, Entity to Author, Identifier1 to AuthorID [also change the data type to Int32] and Message to AuthorName.

Let's add a "Type Descriptor". Right click the *Author* and click on "Add Type Descriptor". Rename it to *ContactNo*. Repeat the steps for City and Country as well. The final output should look similar to Figure 18:

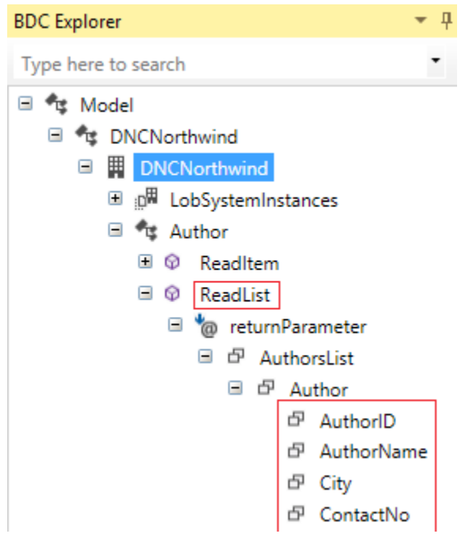


Figure 18 – BDC Explorer

Click on Author and change the type from the properties window as shown in Figure 19:

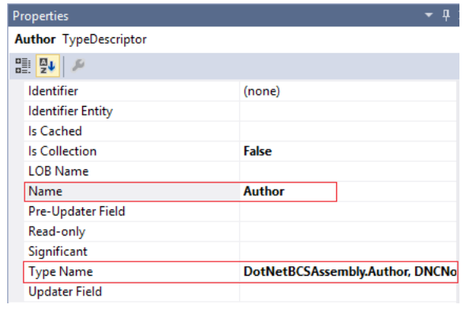


Figure 19 – Type Descriptor Properties

Select *AuthorsList* and change the type name to *IEnumerable<Author>* from the properties window. Repeat the same steps for *ReadItem* method. You can copy and paste the Author type.

Now open *AuthorService* class and write the following code:

```
public class AuthorService
{
    public static Author ReadItem(int id)
    {
        DotNetCurryDataContext dataContext = new
        DotNetCurryDataContext("Data Source=localhost;
        Initial Catalog=DotNetCurryDatabase;
        Integrated Security=True");
        var query = from auth in dataContext.Authors
                    where auth.AuthorID == id
                    select auth;
        return query.SingleOrDefault();
     }
     public static IEnumerable<Author> ReadList()
     {
        DotNetCurryDataContext dataContext = new
        DotNetCurryDataContext("Data Source=localhost;
        Initial Catalog=DotNetCurryDatabase;Integrated
        Security=True");
        var query = from auth in dataContext.Authors
                    select auth;
        return query;
      }
}
```

The last step is to configure the site URL for deployment. Go to Solution Explorer and open Feature1.Template.xml file and write the following code under the *Feature* element –

```
<Properties>
   <Property Key="SiteUrl" Value="http://spspc:19724/
sites/test/"/>
</Properties>
```

Now deploy your solution and set the permissions to your BDC model from the Central Administration tool of SharePoint. Open SharePoint Central Administration tool and click on "Manage Service Applications" located under Application Management and Service Applications as shown in Figure 20:

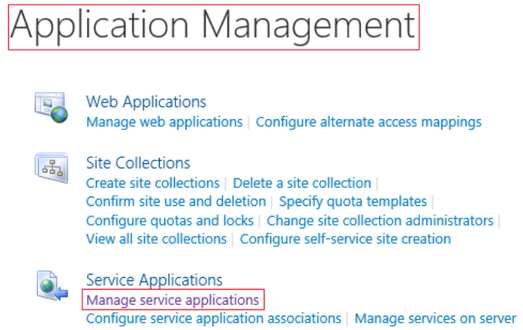


Figure 20 – Manage Service Application

You can view all the service applications. Click on Business Data Connectivity Service and it will show you all the ECTs which you have deployed. Click on Author drop down and set the permission as shown in Figure 21:

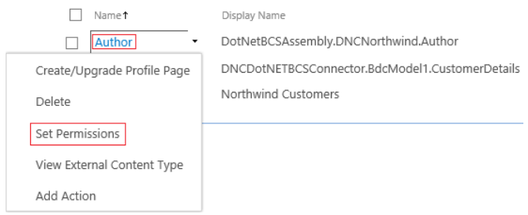


Figure 21 – Setting Permissions to Content Type

Add the user and set the permissions as per your requirement for the user. Now we will create an External List which will make use of our ECT to display the data. Go back to your SharePoint Site and click on Site Contents and then add an app. Select "External List", give it a name and then select Author ECT as shown in Figure 22:

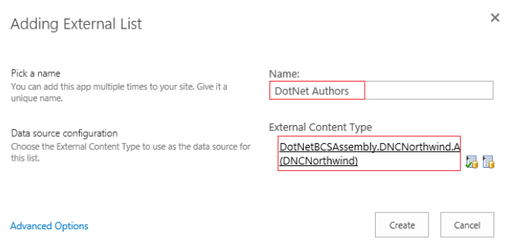


Figure 22 – Adding External List

The output is shown in Figure 23:

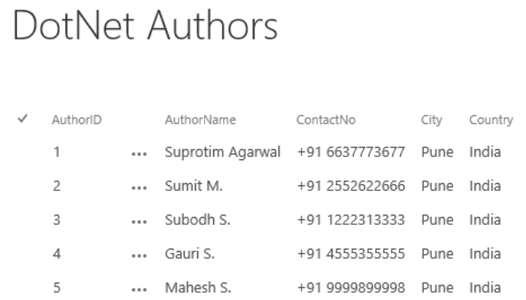


Figure 23 - .NET Connector output

To perform Insert, Update and Delete operations, open the .bdcm file and click on Author entity. Now open BDC Method details window and add the following methods in Figure 24 - Creator, Updater and Deleter.

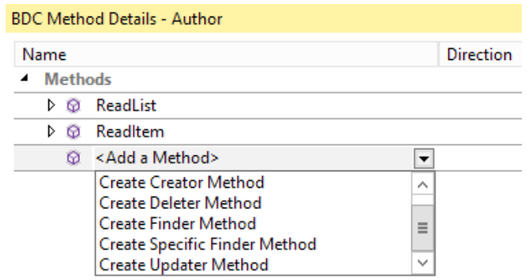


Figure 24 – Insert/Update/Delete BDC Methods

Open AuthorService class and write the code for Insert/Update/Delete methods. The code is shown here –

```
public static Author Create(Author newAuthor)
{
    DotNetCurryDataContext dataContext = new
       DotNetCurryDataContext("Data Source=localhost;
    Initial Catalog=DotNetCurryDatabase;Integrated
    Security=True");
    dataContext.Authors.InsertOnSubmit(newAuthor);
    dataContext.SubmitChanges();
    Author auth = dataContext.Authors.Single
    (a => a.AuthorID == newAuthor.AuthorID);
    return auth;
}
public static void Update(Author author)
{
        DotNetCurryDataContext dataContext = new
        DotNetCurryDataContext("Data Source=localhost;
        Initial Catalog=DotNetCurryDatabase;Integrated
        Security=True");
        var oldAuthor = (from auth in
        dataContext.Authors where auth.AuthorID ==
        author.AuthorID select auth).SingleOrDefault();
        oldAuthor.AuthorName = author.AuthorName;
        oldAuthor.ContactNo = author.ContactNo;
        oldAuthor.City = author.City;
        oldAuthor.Country = author.Country;
        dataContext.SubmitChanges();
}

public static void Delete(int authorID)
{
        DotNetCurryDataContext dataContext = new
        DotNetCurryDataContext("Data Source=localhost;
        Initial Catalog=DotNetCurryDatabase;Integrated
        Security=True");
        var oldAuthor = (from auth in
        dataContext.Authors where auth.AuthorID ==
        authorID select auth).SingleOrDefault();
        dataContext.Authors.DeleteOnSubmit(oldAuthor);
}
```

Now deploy the project and test the ECT by performing CRUD operations. For example, the insert operation should look similar to what is shown in Figure 25:

Figure 25 – Insert Operation in ECT

CREATING BCS CONNECTOR USING WCF SERVICE

We will now take a look at the WCF Connector. We will create a WCF Service and host it on IIS. Open Visual Studio and create a WCF Service Application with the name *AuthorService*. VS will create an IService1 interface. Add the following code in this interface –

```
[ServiceContract]
    public interface IService1
    {
        [OperationContract]
        List<Author> ReadList();
        [OperationContract]
        Author ReadItem(int AuthorID);
        [OperationContract]
        Author Create(Author newAuthor);
        [OperationContract]
        void Update(Author newAuthor);
        [OperationContract]
        void Delete(int AuthorID);
    }
```

Implement this interface in Service1 class. Then add a LINQ to SQL class with the name DotNetCurry and write the following code in our Operation contract –

```
[AspNetCompatibilityRequirements(RequirementsMode=
AspNetCompatibilityRequirementsMode.Allowed)]
public class Service1 : IService1
{
DotNetCurryDataContext dataContext =
new DotNetCurryDataContext();
public List<Author> ReadList()
{
    var query = from auth in dataContext.Authors
                select auth;
    return query.ToList();
}
public Author ReadItem(int AuthorID)
{
    var query = (from auth in dataContext.Authors
                 where auth.AuthorID==AuthorID
                 select auth).SingleOrDefault();
    return query;
}
public Author Create(Author newAuthor)
{
    dataContext.Authors.InsertOnSubmit(newAuthor);
    dataContext.SubmitChanges();
    Author auth= dataContext.Authors.Single(
     a => a.AuthorID == newAuthor.AuthorID);
    return auth;
}
public void Update(Author author)
{
    var oldAuthor = (from auth in dataContext.
                     Authors where auth.AuthorID ==
                     author.AuthorID
                      select auth).SingleOrDefault();
        oldAuthor.AuthorName = author.AuthorName;
        oldAuthor.ContactNo = author.ContactNo;
        oldAuthor.City = author.City;
        oldAuthor.Country = author.Country;
        dataContext.SubmitChanges();
}
public void Delete(int AuthorID)
{
    var oldAuthor = (from auth in dataContext.Authors
                      where auth.AuthorID == AuthorID
                      select auth).SingleOrDefault();
    dataContext.Authors.DeleteOnSubmit(oldAuthor);
    dataContext.SubmitChanges();
}
}
```

Once our service is ready, publish the same on IIS. Now open the site in SharePoint Designer. I am using SharePoint Designer 2013. In the left hand Navigation pane, click on *External Content Types* and click on External Content Type from ribbon. Name the External Content Type as *WCF Authors*. Click on the link to discover the external data sources.

Now select the WCF Service Connector and set the WSDL URL and End point address to the one shown in Figure 26:

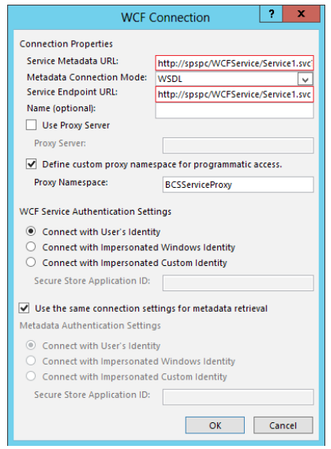


Figure 26 – WCF Connection

After a successful configuration, you will see the web methods as shown in Figure 27:

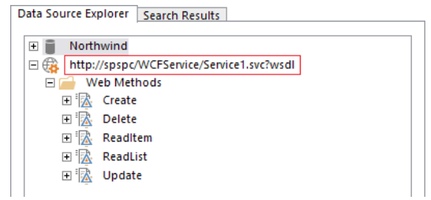


Figure 27 – Data Source Explorer

Now right click the ReadList method and select "New Read List Operation" which brings up a wizard. Follow the steps of the wizard. Repeat the same step for all the web methods and save your work.

Also make sure that you will set the appropriate permissions to the content type using SharePoint Central Administrator tool as described in the previous steps.

Now go back to the SharePoint Site and create an External List as described in previous steps. The list is displayed in Figure 28:

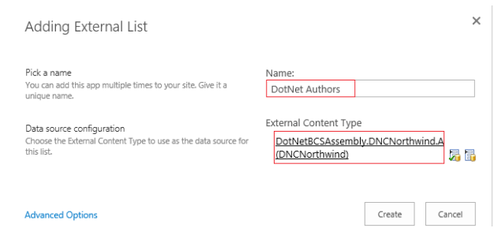


Figure 28 – Adding External List

The output of this task is shown in Figure 29:

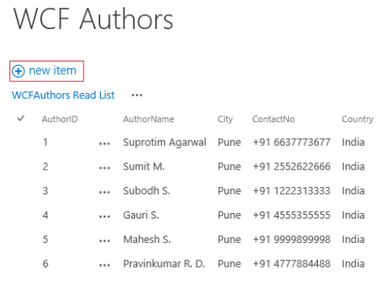


Figure 29 – Data Displayed using WCF

When you click on the *new item* link, you will see the following form shown in Figure 30. Fill the details and add a new item to the list.

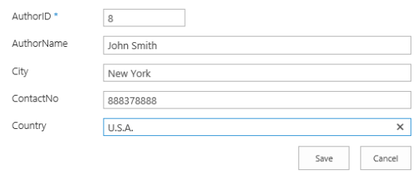


Figure 30 – Insert using WCF

CREATING AN ODATA SERVICE TO FETCH DATA FROM DATABASE

OData Service is new to SharePoint 2013. We will first design an OData service which will fetch the data from Northwind database which I assume is available in your SQL Server.
To create an OData service, open Visual Studio 2012/2013 and create an Empty Web Application with the name "NorthwindODataService". Then right click the web project and add a new Item. From the Visual C# section located on the left side, choose "Data" and then choose the ADO.NET Entity Data Model. Clicking on the "Add" button brings up the Entity Data Model Wizard. Choose *Generate* from the database. Configure the connection string to your Northwind database and click on the "Next" button.

In this step, we will choose the Entities. Expand the Tables section and from the dbo schema choose "Customers", "Orders", "Order Details", "Employees" and "Products". Once you choose these tables, click on the *Finish* button. Your Entity Model should look as shown in Figure 31:

Figure 31 - Entity Model

Now since our data model is ready, let's create an OData service which will fetch data from the Northwind database using our Entity Model. Right click the web project and add a New Item. Select WCF Data Service and name it as "NWService.svc". Change the code to the following:

```
namespace NorthwindODataService
{
    public class NWService : DataService<NorthwindEntities>
    {
        public static void InitializeService(DataServiceConfiguration config)
        {
            config.SetEntitySetAccessRule("*", EntitySetRights.All);
            // config.SetServiceOperationAccessRule("MyServiceOperation",
ServiceOperationRights.All);
            config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V3;
        }
    }
}
```

Once the service is configured, you can test the service. Right click the NWService.svc file and click on "View in Browser". You will see all the Entities which we chose earlier. Try changing the URL to the following and try out some queries:

1. http://localhost:3025/NWService.svc/Customers
2. http://localhost:3025/NWService.svc/Customers('ALFKI')
3. http://localhost:3025/NWService.svc/Customers('ALFKI')/Orders

Now it's time to deploy your service under IIS. You can deploy your OData service in many ways. I am going to deploy the OData service using manual deployment. Open Internet Information Services (IIS) Manager. Create a blank folder under C:\inetpub\wwwroot\OData folder with the name OData. Create a new web site with the name *DNCODataServices* and map the above OData folder to the Web site.

Then copy all the required files from your NorthwindODataService web project and paste those files under "C:\inetpub\wwwroot\OData". Now the most important thing is to set the *Identity* of the Application Pool which got created when we created our Web site. This is shown in Figure 32:

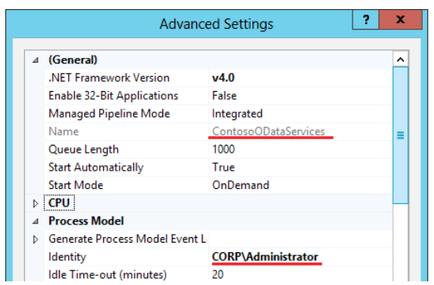


Figure 32 – Application Pool Settings

Fetching External Data Into SharePoint using OData

To fetch External Data into SharePoint, we will now design a BDC Model which will make use of our OData service as a data source to fetch data from the Northwind database. This returns a big XML document that we will not reproduce here, but we will look at it briefly in portions. Please download the entire BDC file from the code download at the end of this article and upload the file as described in Figure 33.

Here are some steps and points to consider:

1. The Model name is "DNCNorthwindMetadata".
2. The data source of this Model is OData.
3. We are specifying ODataServiceMetadataUrl which is "http://localhost:8888/NWService.svc/$metadata". Please note that this the service which we have hosted in IIS in our previous steps.
4. The Authentication mode is set to "PassThrough".

```
<?xml version="1.0" encoding="UTF-16"?>
- <Model xmlns="http://schemas.microsoft.com/windows/2007/BusinessDataCatalog"
Name="DNCNorthwindMetadata" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  - <LobSystems>
    - <LobSystem Name="DNC" Type="OData">
      - <Properties>
          <Property Name="ODataServiceMetadataUrl"
            Type="System.String">http://localhost:8888/NWService.svc/
            $metadata</Property>
          <Property Name="ODataServiceMetadataAuthenticationMode"
            Type="System.String">PassThrough</Property>
          <Property Name="ODataServicesVersion"
```

5. The second section is the LOB System Instances. The name of our LOB System Instance is DNC Northwind.
6. The OData Service URL is "http://localhost:8888/NWService.svc".
7. The authentication mode is Pass Through.
8. The OData format to expose the data is **application/atom+xml**.

```
- <LobSystemInstances>
  - <LobSystemInstance Name="DNCNorthwind">
    - <Properties>
        <Property Name="ODataServiceUrl"
          Type="System.String">http://localhost:8888/NWService.svc</Property>
        <Property Name="ODataServiceAuthenticationMode"
          Type="System.String">PassThrough</Property>
        <Property Name="ODataFormat"
          Type="System.String">application/atom+xml</Property>
        <Property Name="HttpHeaderSetAcceptLanguage"
          Type="System.Boolean">true</Property>
      </Properties>
    </LobSystemInstance>
  </LobSystemInstances>
```

9. The third section is the specification of Entities and their methods like –
   a. Reading all Customers
   b. Read Specific Customer
   c. Insert New Customer
   d. Update Customer
   e. Delete Customer

10. In the above model, we have the following methods with their Parameters –
    f. Read All Customers. The URL is –

```
<Property Name="ODataEntityUrl" Type="System.String">/Customers?
    $top=@LimitCustomers</Property>
```

    g. Read Specific Customer. The URL is –

```
<Properties>
    <Property Name="ODataEntityUrl" Type="System.String">/Customers
        (CustomerID='@CustomerID')</Property>
```

    h. Create Customer. The URL is –

```
<Properties>
    <Property Name="ODataEntityUrl" Type="System.String">/Customers
        (CustomerID='@CustomerID')</Property>
```

    i. Update Customer. The URL is –

```
<Properties>
    <Property Name="ODataEntityUrl" Type="System.String">/Customers
        (CustomerID='@CustomerID')</Property>
```

    j. Delete Customer. The URL is –

```
<Properties>
    <Property Name="ODataEntityUrl" Type="System.String">/Customers
        (CustomerID='@CustomerID')</Property>
```

Once your Model is ready, let's import it into our Business Connectivity Services using SharePoint Central Administration Tool. To import the model, open Central Administration and click on *Application Management* link which is available in the Left Navigation Pane. Under Application Management > Service Applications > click on Manage Service Applications.

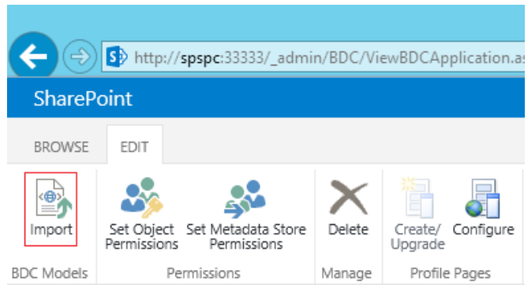


Figure 33 – Import BDC Model

Click on "Business Data Connectivity Service". In the "Edit" ribbon, click on "Import" button and browse the BDC Model file as shown in Figure 33. This will import the file into Business Data Connectivity Service. Now the most important part is to set the permissions to the entities of our BDC Model. Once the permissions are set, the model is ready to use. This is shown in Figure 34:

	CustomerID		CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country
	ALFKI	...	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin	DC	12209	Germany
	ANATR	...	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222	México D.F.		05021	Mexico
	ANTON	...	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.		05023	Mexico
	AROUT	...	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London		WA1 1DP	UK

Figure 34 – Set BDC Model Permission

Now let's use the BDC model which we just imported in our SharePoint site. We will create an External Content Type in our SharePoint Site. Let's go back to our SharePoint site and click on "Site Contents" and then click on "add an app". Choose External Site and configure it by giving it a name "OData Customers" and select the DNC Northwind Customers Content type.

The output of the "OData Customers" list is shown in Figure 35:

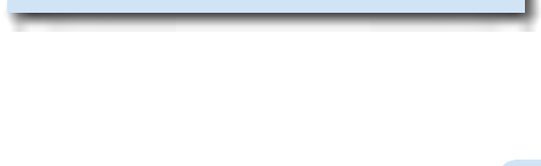


Figure 35 – OData External List

CONCLUSION

In this article, we saw four different BCS connectors using which we can connect to external systems and create a centralized data source in SharePoint. We saw the security features of BCS and how SharePoint 2013 has been enhanced to support OData Services for accessing data via BCS. We also saw how External Content Types allows for SharePoint applications to access centralized data stores and perform CRUD operations ■

Pravinkumar works as a freelance corporate trainer and consultant on Microsoft Technologies. He is also passionate about technologies like SharePoint, ASP.NET, WCF. Pravin enjoys reading and writing technical articles. You can follow Pravinkumar on Twitter @pravindotnet and read his articles at bit.ly/pravindnc

Windows Phone

> THE NOKIA IMAGING SDK IS A PRODUCTIVE LIBRARY FOR MANIPULATING IMAGES IN AN EFFICIENT WAY IN WINDOWS PHONE 8.X AND WINDOWS 8.1

BUILDING AN IMAGE TRANSFORMATION APPLICATION FOR WINDOWS PHONE 8

The mobile phone industry is highly competitive. Microsoft realizes this and has been taking huge initiatives to improve the User and Developer experience for the Windows Phone Platform. The acquisition of Nokia Devices and Services business is a part of this initiative. Nokia continues to capture the Windows Phone market like no other OEM's and continues to deliver excellent devices and a rich phone development platform.

Nokia has an edge over other Mobile vendors in their rich set of API's and the documentation that comes along with it. Nokia Imaging SDK is one of these platforms that developers can leverage to deliver some cool imaging experiences to consumers. Other popular API's by Nokia are HERE Maps and Nokia MixRadio.

In this article, we will make use of the Imaging SDK and build an Image filtering application which will allows us to apply various filters and effects to our Images and also enables us to save and share them on Social platform.

WHAT IS NOKIA IMAGING SDK?

Nokia Imaging SDK consists of APIs which allows developers to apply various filters and effects to images either taken from the Camera or on images which already exists on the phone. This SDK also allows advance image operations like Lens Blur, Cropping etc. and allows you to incorporate your own custom filters as well. In this article we are using Nokia Imaging SDK 1.1. Recently Nokia also announced version 1.2 but since it is still in Beta (as of this writing), we have focused on SDK 1.1.

CREATING A NEW WINDOWS PHONE 8.0 APPLICATION

We will be building an application for the Windows Phone 8.0 Platform. Use a New Windows Phone Blank Project in Visual Studio 2013, to start with

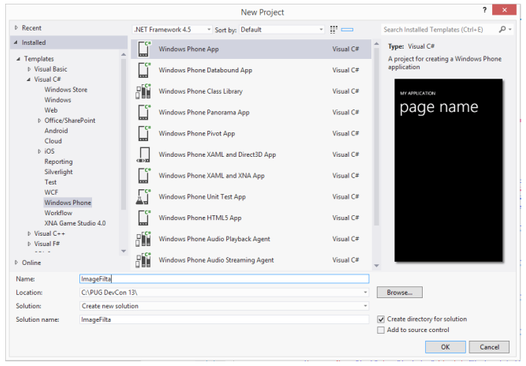


Figure 1: Windows Phone Project Template

We are referring this application by the name "ImageFilta", so you may see "ImageFilta" used throughout the article.

NOTE: *If you use this application directly in Windows Phone 8.0 then it works as-is. You may have to change code in case if you choose Windows Phone 8.1 Blank App (Silverlight) Project Template. For Windows Phone 8.1 Blank App, which is WinRT based template, most of the code blocks will not work. So it is highly recommended that you rewrite this application in Windows Phone 8.1 if you are planning to migrate.*

Install Nokia Imaging SDK version 1.1

To download the Nokia Imaging SDK from Visual Studio 2013, you can go to your NuGet package manager or package manager console and install the APIs as shown below:

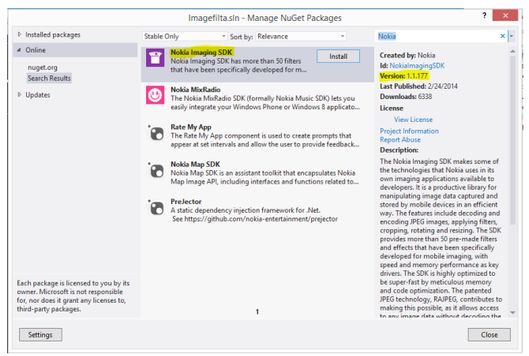


Figure 2: Nokia Imaging SDK NuGet Package

OR

```
PM> Install-Package NokiaImagingSDK -Version 1.1.177
```

Figure 3: Package Manager Console

The URL to download Nokia Imaging SDK is http://www.nuget.org/packages/NokiaImagingSDK/1.1.177

Once you install the package, you just need to use the libraries and start using the APIs. There are no registration or key or configuration settings like we have in the case of Maps application. However it is very important to note that while building app using Nokia Imaging SDK, you need to explicitly delete the "Any CPU" from Configuration Manager dialog in Visual Studio > Active Solution Platform and Keep x86. If you do not delete "Any CPU", exception will be thrown while you execute your application.

NOTE: *If you are interested to read more about Any CPU, Rick Byers offers a good explanation on why AnyCPU are usually more trouble than they're worth.*

BUILDING UI FOR IMAGEFILTA

What we are looking to build is a page with various filters and options. There is a ScrollViewer at the bottom which will allow us to scroll across filters. This ScrollViewer contains a set of Image Controls. When a user taps the image, we will render a filtered image above it and perform operations like Save or

Share from the AppBar at Bottom. This is how the UI will look like

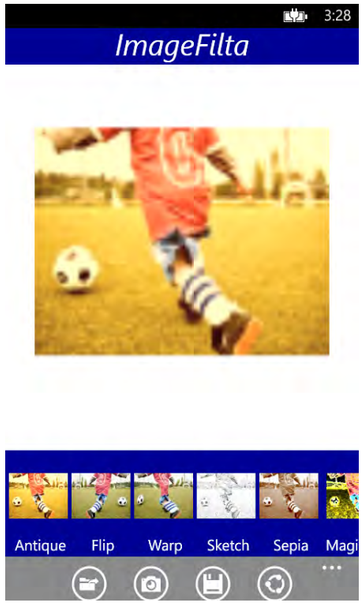


Figure 4: ImageFilta UI with Filters

Here is the XAML for above UI

```
<phone:PhoneApplicationPage.Resources>
<Style TargetType="Image">
    <Setter Property="Width" Value="80"></Setter>
    <Setter Property="Height" Value="80"></Setter>
</Style>

<Style TargetType="Image" x:Key="img">
    <Setter Property="Width" Value="400"></Setter>
    <Setter Property="Height" Value="400"></Setter>
</Style>
</phone:PhoneApplicationPage.Resources>
<Grid x:Name="LayoutRoot" Background="White">
<Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>

<!--TitlePanel contains the name of the application and
page title-->
<StackPanel x:Name="HeaderSection" Grid.Row="0"
Margin="0,0,0,28" Background="DarkBlue">
    <TextBlock HorizontalAlignment="Center"
    Text="ImageFilta"
        FontSize="40" FontFamily="Segoe Script"
        FontStyle="Italic"  />
</StackPanel>

<!--ContentPanel - place additional content here-->
<Grid x:Name="ContentPanel" Grid.Row="1"
Margin="0,0,0,0" Background="White">

<Image VerticalAlignment="Top" Margin="10,5,10,0"
x:Name="imgOrigin" Style="{StaticResource img}"
Stretch="UniformToFill"/>

<ScrollViewer VerticalAlignment="Bottom"
Margin="0,0,0,-7" Height="155"
HorizontalScrollBarVisibility="Auto"
VerticalScrollBarVisibility="Disabled">

<StackPanel Orientation="Horizontal" Height="140"
x:Name="stkfilters" Background="DarkBlue">
    <StackPanel>
        <Image x:Name="Img1" Tap="Img1_Tap"
        Stretch="UniformToFill"
        Margin="5,20,0,0" />
        <TextBlock HorizontalAlignment="Center"
        Margin="10,10,0,0"
        Text="Antique"></TextBlock>
    </StackPanel>
    ...
    <StackPanel>
        <Image x:Name="Img10" Tap="Img10_Tap"
        Stretch="UniformToFill" Margin="5,20,0,0"/>
        <TextBlock HorizontalAlignment="Center"
        Margin="10,10,0,0" Text="Stamp"></TextBlock>
    </StackPanel>
</StackPanel>
</ScrollViewer>
</Grid>
</Grid>
```

XAML for App Bar :

```
<phone:PhoneApplicationPage.ApplicationBar >
<shell:ApplicationBar Mode="Default" Opacity="1.0"
IsMenuEnabled="True" IsVisible="True"
BackgroundColor="Gray">

<shell:ApplicationBarIconButton x:Name="mnuOpen"
IconUri="/Images/Open.png" Text="album"
Click="mnuOpen_Click"/>

<shell:ApplicationBarIconButton x:Name="mnuCamera"
IconUri="/Images/Camera.png" Text="camera"
Click="mnuCamera_Click"/>

<shell:ApplicationBarIconButton x:Name="mnuSave"
IconUri="/Images/Save.png" Text="save"
Click="mnuSave_Click"/>

<shell:ApplicationBarIconButton x:Name="mnuShare"
IconUri="/Images/Share.png" Text="share"
Click="mnuShare_Click"/>

    <shell:ApplicationBar.MenuItems>
        <shell:ApplicationBarMenuItem Text="Send
        Feedback" />

        <shell:ApplicationBarMenuItem Text="About Us" />
    </shell:ApplicationBar.MenuItems>

</shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

Now let's talk about each functionality one by one. Before that let's get familiar with AppBar options here

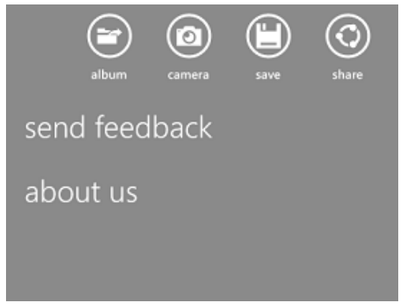


Figure 5: AppBar Options

As you can see from the AppBar, there are four Primary options:

1. Load Image from Existing Storage(Phone/Albums)
2. Capture Live image from Camera
3. Save Image after applying specific filter to JPEG format
4. Share Image on Socials available on phone.

Now let's get into more details for each functionality.

LOAD IMAGE FROM EXISTING STORAGE

In this option, we will look at the approach to select an Image from the existing Albums on the Phone. For this purpose, we will make use of the *PhotoChooserTask*. Once you click on the "Open" icon in the AppBar, it will redirect you to Photos/Albums section on the phone.

```
// check namespace used from the code download

#region Global Variables
    private string _fileName = string.Empty;
    private String filtername = string.Empty;
    private StreamImageSource source;
#endregion

private void mnuOpen_Click(object sender, EventArgs e)
{
    PhotoChooserTask chooser = new PhotoChooserTask();
    chooser.Completed += SelectImage;
    chooser.Show();
}
```

We are using the *PhotoChooserTask* which comes out of the box in Windows Phone 8.0 SDK. It allows us to navigate through albums and select individual images. You can see that we have attached a method "SelectImage" to chooser.Completed event. We will look at the "SelectImage" method shortly after looking at the approach of fetching live images from Camera.

TAKING LIVE IMAGES FROM THE CAMERA

To capture live image from the Camera, we will use *CameraCaptureTask* available in Windows Phone 8.0 SDK out of the box. CameraCaptureTask will also allow us to accept or reject any image taken from the Camera as shown in Figure 6:

Figure 6: Accept Retake options

```
private void mnuCamera_Click(object sender, EventArgs
e)
{
    CameraCaptureTask cmrcapture = new
    CameraCaptureTask();
    cmrcapture.Completed += SelectImage;
    cmrcapture.Show();
}
```

Now let's talk about *SelectImage* which is a common method for both the existing images and live images options. The SelectImage method allows you to apply filter on the fly to the selected image, either from Album or from Camera.

```
private async void SelectImage(object sender,
PhotoResult e)
{
if (e.TaskResult != TaskResult.OK || e.ChosenPhoto ==
null)
{
    MessageBox.Show("Invalid Operation");
    return;
}

try
{
    stkfilters.Visibility = System.Windows.Visibility.
    Visible;

    source = new StreamImageSource(e.ChosenPhoto);
    // Antique Filter
    using (var antiquefilters = new FilterEffect(source))
    {
        var antique = new AntiqueFilter();
        antiqufilters.Filters = new IFilter[] { antique
        };

        var target = new WriteableBitmap((int)Img1.
        ActualWidth, (int)Img1.ActualHeight);

        using (var renderer = new
    WriteableBitmapRenderer(antiqufilters, target))
    {
        await renderer.RenderAsync();
        Img1.Source = target;
    }
}

//Flip Filter
using (var flipfilters = new FilterEffect(source))
{
    var flip = new FlipFilter(FlipMode.Horizontal);
    flipfilters.Filters = new IFilter[] { flip };

    var target = new WriteableBitmap((int)Img2.
    ActualWidth, (int)Img2.ActualHeight);

    using (var renderer = new
    WriteableBitmapRenderer(flipfilters, target))
    {
        await renderer.RenderAsync();
        Img2.Source = target;
    }
}

    }

    //Wrap Filter
    ...
    //Sketch Filter
    ...
    //Sepia Filter
    ...
    //MagicPen Filter
    ...
    //Solarize Filter
    ...
    //Negative Filter
    ...
    //Temparature and Tint Filter
    ...
    //Stamp Filter
    ...
}

catch (Exception exception)
{
    MessageBox.Show("Some error occured, Please share
    feedback from Send Feedback Menu" + exception.
    Message.ToString());
        return;
}
}
```

You can see that "SelectImage" takes *PhotoResult* as a parameter. Since we are taking an image from the Camera as well, it is recommended to use *StreamImageSource*. We are applying filters to individual images at the bottom, which shows the preview of the filter. We are also sending a couple of parameters to the filters like for StampFilter, TemperatureAndTintFilter etc. To know more about an individual Filter, you can visit the Nokia Imaging SDK API Documentation at

http://developer.nokia.com/resources/library/Imaging_API_Ref/nokiagraphicsimaging-namespace.html

You can see how these filters get applied to the individual images as shown here:

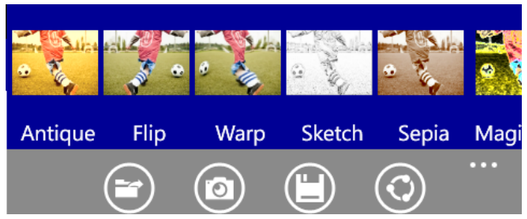


Figure 7: Filters applied to images

To see a larger view of the image, click on the individual image containing that filter effect:

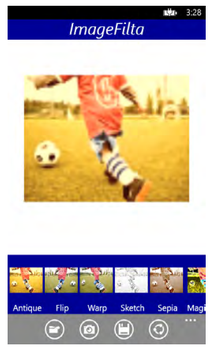


Figure 8: Antique Filter been applied

This can be achieved by tapping on each image as shown in the code here:

```
private void ApplyToOriginal(Image img)
{
    imgOrigin.Source = img.Source;
}

private void Img1_Tap(object sender, System.Windows.
Input.GestureEventArgs e)
{
    ApplyToOriginal(Img1);
    filtername = "Antique";
}

private void Img2_Tap(object sender, System.Windows.
Input.GestureEventArgs e)
{
    ApplyToOriginal(Img2);
       filtername = "Flip";
}
```

Similarly you can follow the same procedure for the rest of the eight Images. After applying filters, we need to save our modified image to the Phone. For this, we will write a helper class "SaveImageHelper.cs"

SAVING IMAGE WITH FILTER EFFECT TO THE PHONE

To save an image, we are assigning *Filtername* to each image in the ScrollViewer where we can see a list of images having the filter effects. So as discussed earlier, we will pass the *filtername* as a parameter along with filter and Image to the method in "SaveImageHelper.cs". We have taken a simple Switch..Case here since we have limited number of filters. For our application we are not following any structure design pattern but you can always leverage some design patterns like MVVM.

```
private void mnuSave_Click(object sender, EventArgs e)
{
SaveImageHelper smhelp = new SaveImageHelper();
try
{
    switch (filtername)
    {
        case "Antique": AntiqueFilter ant = new
              AntiqueFilter();
              smhelp.SaveImage(source, Img1,
              _fileName, ant);
```

```
            break;

        case “Flip”: FlipFilter flip = new
            FlipFilter(FlipMode.Horizontal);
            smhelp.SaveImage(source, Img2,
            _fileName, flip);
            break;
            ...

        default:
            break;
        }
}
catch (Exception exception)
{
    MessageBox.Show(“Error Occured, Unable to Save
    Picture” + exception.Message.ToString());
}
}
```

As shown in the code, we are using the *SaveImage()* method from “SaveImageHelper.cs”. Here is the implementation of the SaveImage() Method.

```
public async void SaveImage(StreamImageSource source,
Image Img, String FilterName,IFilter Filter)
{
    using (var imgfilters = new FilterEffect(source))
    {
        var filt = Filter;
        imgfilters.Filters = new IFilter[] { filt };

        var target = new WriteableBitmap((int)Img.
        ActualWidth, (int)Img.ActualHeight);
        using (var renderer = new
        WriteableBitmapRenderer(imgfilters, target))
        {
            await renderer.RenderAsync();
            Img.Source = target;
        }

        var jpegRenderer = new JpegRenderer(imgfilters);

        IBuffer jpegOutput = await jpegRenderer.
        RenderAsync();

        MediaLibrary library = new MediaLibrary();
        _fileName = string.Format(“ImageFilta”,
        DateTime.Now.Second) + “.jpg”;
        var picture = library.SavePicture(_fileName,
        jpegOutput.AsStream());
        MessageBox.Show(“Image saved!”);
    }
}
```

Once you click on *Save* from the AppBar, the image will be saved on your phone and you can check the same in your album.

Figure 9: Image saved on your phone

SHARING IMAGE WITH FILTER EFFECT ON SOCIALS

Social integration is one of the key aspects of this application since the very purpose of this application is to allow users to apply some filters and effects to the images and share them with their friends on the social network. The following code shows how it can be done quickly. Note that this feature totally depends on the Social platform integration configured on your device. It does not explicitly post anywhere else, unless you select to do so.

```
private void mnuShare_Click(object sender, EventArgs e{
    var chooser = new Microsoft.Phone.Tasks.
    PhotoChooserTask();
    chooser.Completed += chooser_Completed;
    chooser.Show();
}

void chooser_Completed(object sender, PhotoResult e){
    _fileName = e.OriginalFileName.ToString();
    if (e.TaskResult == TaskResult.OK)
    {
        ShareMediaTask shareMediaTask = new
        ShareMediaTask();
        shareMediaTask.FilePath = _fileName;
        shareMediaTask.Show();
    }
}
```

The application simply makes use of *Launchers* and *Chooser* throughout the application. To allow users to select a photo, the *PhotoChooserTask* class is used and to allow the app to launch a dialog that allows a user to share a media file on the social networks of their choice, the *ShareMediaTask* class is used. These classes are available in Windows Phone 8.0 SDK out of the box. So this is how you can pick images from Saved Picture and share on Socials sites.

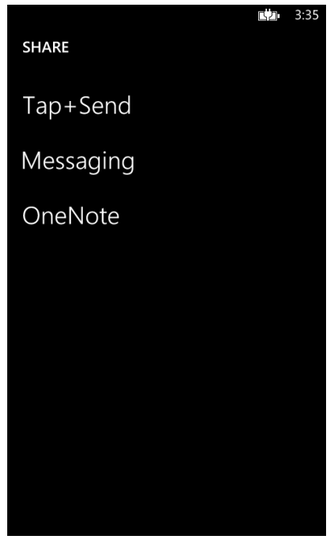


Figure 10: Social Sharing Features

SUMMARY

There are some Image filtering apps available on the Windows Phone Store like Instagram, 6tag etc. This article has been written with the purpose of guiding you and giving you a roadmap of how such apps can be built using Nokia Imaging SDK. Nokia Imaging SDK is free to download, however if you are using it in your commercial / store ready app, then please check the Terms and Conditions mentioned on the Nokia Imaging SDK Website.

Nokia Imaging SDK provides a simple and easy way to implement APIs and thus makes it easy for developers to plug and play in their apps. We all know that Nokia as a Windows Phone manufacturer comes with an awesome set of Hardware devices which also include a stunning Camera built-into these devices. Nokia Devices like Lumia 920, 1520 and the 1020 in particular, which gives a 41 megapixel camera, are some of the popular Windows Phone devices. In order to enjoy the imaging from such devices, there is always need of image editing and processing. Nokia Imaging SDK gives you access to a powerful library of exciting image-manipulation tools ■

Download the entire source code from our GitHub Repository at bit.ly/dncm13-wpnokiasdk

Vikram Pendse is currently working as a Subject Matter Expert in Avanade (Accenture, India) for .NET, Azure & Windows Phone Projects. He is responsible for Providing Estimates, Architecture, Supporting RFPs and Deals. He is Microsoft MVP since year 2008 and currently a Windows Phone MVP. He is a very active member in various Microsoft Communities and participates as a “Speaker” in many events. You can reach him at: vikrampendse@hotmail.com or Follow on Twitter: @VikramPendse

CHAIN AJAX REQUESTS WITH JQUERY DEFERRED

Imagine a scenario where you have a bunch of functions to execute asynchronously, but each function depends on the result of the previous one. You do not have an idea when each function will complete execution. In such cases, you can write Callbacks. Callbacks are useful when working with background tasks because you don't know when they will complete. Callbacks can be defined as:

```
function A(callback) {
    $.ajax({
        //...
        success: function (result) {
            //...
            if (callback) callback(result);
        }
    });
}
```

And here's a prototype of callbacks in action:

```
A(function () {
    B(function () {
        C()
    })
});
```

However this code style leads to too much nesting and becomes unreadable if there are too many callback functions. Using jQuery, the same functionality can be achieved elegantly without too much nesting. The answer lies in *Deferred* and *Promise*. Let's cook up an example:

We have four functions A(), B(), C() and D() that will execute asynchronously. However function B relies on the result of function A(). Similarly function C relies on the result of function B and so on. The task is to execute these functions one after the other and also make the results of the previous function available in the next one.

> Asynchronous requests cannot be guaranteed to finish in the same order that they are sent. However using deferred objects, we can make sure that the *callbacks* for each async request runs in the required order

Observe this code:

```
function A() {
    writeMessage("Calling Function A");
    return $.ajax({
        url: "/scripts/S9/1.json",
        type: "GET",
        dataType: "json"
    });
}

function B(resultFromA) {
    writeMessage("In Function B. Result From A = " +
    resultFromA.data);
    return $.ajax({
        url: "/scripts/S9/2.json",
        type: "GET",
        dataType: "json"
    });
}

function C(resultFromB) {
    writeMessage("In Function C. Result From B =  " +
    resultFromB.data);
    return $.ajax({
        url: "/scripts/S9/3.json",
        type: "GET",
        dataType: "json"
    });
}

function D(resultFromC) {
    writeMessage("In Function D. Result From C = " +
    resultFromC.data);
}

A().then(B).then(C).then(D);

function writeMessage(msg) {
    $("#para").append(msg + "</br>");
}
```

Observe this important piece of code

```
A().then(B).then(C).then(D);
```

From the jQuery Documentation: *Callbacks are executed in the order they were added. Since deferred.then returns a Promise, other methods of the Promise object can be chained to this one, including additional .then() methods.*

And that's what we are doing here. Every Ajax method of jQuery already returns a *promise*. When the Ajax call in function A completes, it *resolves* the promise. The function B() is then called with the results of the Ajax call as its first argument.

When the ajax call in B() completes, it resolves the promise and function C() is called with the results of that call and so on. Here we are just adding *return* in every function to make this chain work.

By the way, you may have observed that we are repeating settings for multiple AJAX calls in the same script. You can clean up the code further by specifying global settings using `$.ajaxSetup`. Here's how:

```
$.ajaxSetup({
  type: 'GET',
  dataType: "json",
  delay: 1
});
```

and then each call is reduced to

```
function A() {
    writeMessage("Calling Function A");
    return $.ajax({
        url: "/scripts/S9/1.json",
    });
}
```

Run the sample and you will get the following output: depending on what's inside 1.json. 2.json and so on.

```
Calling Function A
In Function B. Result From A = 1
In Function C. Result From B = 2
In Function D. Result From C = 3
```

Once you've got your head wrapped around them, jQuery Deferred can be simple yet powerful. The ability to easily chain methods is beautiful and easy on the eyes!

This article was taken from my upcoming ***The Absolutely Awesome jQuery Cookbook*** at www.jquerycookbook.com ■

Suprotim Agarwal, ASP.NET Architecture MVP, is an author and the founder of popular .NET websites like dotnetcurry.com, devcurry.com and the DNC .NET Magazine that you are reading. You can follow him on twitter @suprotimagarwal or check out his new book www.jquerycookbook.com

Getting started with

ASP.NET vNext and Visual Studio 2014 CTP: The Bright Future

Editor's Note: .NET vNext can be termed as the next generation of .NET. Although this is an early preview release (call it a v0.1 if you may), it does gives us some very interesting insights into what's cooking in Redmond.

ASP.NET vNext has been completely refactored and is no more tied to the .NET Framework. This allows us to bundle and deploy our own version of the .NET framework on an app-by-app basis. What this means is we can update our .NET libraries on the server without breaking our ASP.NET vNext apps, as our apps will continue using the .NET version that was deployed with it, and also give us the flexibility to choose our version, going forward. The idea of running apps that use different versions of .NET, side-by-side, is very cool!

ASP.NET vNext will be optimized for the cloud so you do not have to deploy the entire ASP.NET framework. You only deploy the pieces that you need, making the deployment smaller, more manageable and componentized. Microsoft quotes that "*vNext apps can make use of cloud optimized subset of .NET framework whose approximate size is around 11 megabytes compared to full .NET framework which is around 200 megabytes in size*". Now that's something!

You will also get dynamic compilation available in the new open-source Roslyn .NET compiler platform, for better startup times. Now you no longer have to recompile your apps to see the changes you have made in your browser. Needless to mention that vNext, Roslyn, MVC etc, is open source (http://www.microsoft.com/en-us/openness/default.aspx#projects). So you can start contributing to vNext right away!

One of the biggest and exciting changes in ASP.NET vNext is the cross platform support. Microsoft is actively working with companies like Xamarin, so that you can start hosting your ASP.NET vNext apps on Unix or OS X on top of the Mono runtime. Check out this article by Graeme (http://graemechristie.github.io/graemechristie/blog/2014/05/26/asp-dot-net-vnext-on-osx-and-linux/) where he demonstrates how to run ASP.NET vNext on OS X and Linux. Pretty cool huh!

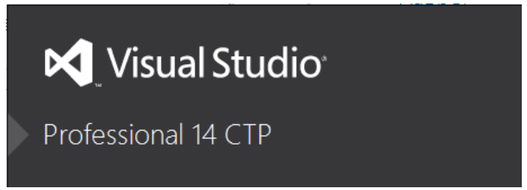
ASP.NET MVC, Web APIs and Web Pages are now merged into one single programming model which is called as MVC 6. Similar to Web API 2 and SignalR 2, you can now self-host vNext app into custom processes. The dependency injection is now built into the framework, but you are allowed to use your own IoC container. For all the NuGet fans, you can NuGet everything, including the runtime.

***Note:** ASP.NET WebForms will run on .NET vNext but not on ASP.NET vNext or the cloud optimized version of ASP.NET vNext.*

Microsoft realizes that its products and the framework have to evolve in order to stay relevant in a world, where apps have to target multiple platforms. At TechEd North America held in May 2014, Microsoft announced ASP.NET vNext, which is built on .NET vNext. The strategy was crystal clear - a mobile-first, cloud-first world!

Although Microsoft has just released the first preview of Visual Studio 2014 CTP, we will look into some cool features introduced in ASP.NET vNext and Visual Studio 2014 CTP. You can download the CTP from here - http://www.visualstudio.com/en-us/downloads/visual-studio-14-ctp-vs. Do not install it side by side with an existing Visual Studio setup, since vNext is still in CTP. Use a virtual box like Hyper-V.

Assuming you're ready to go, fire up Visual Studio 2014 CTP –



Click on New Project and choose ASP.NET vNext Web Application as shown in Figure 1:

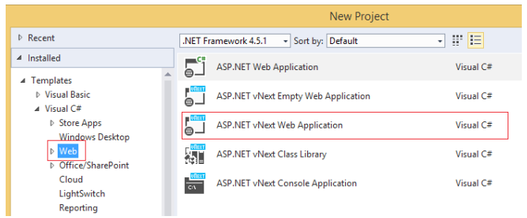


Figure 1: ASP.NET vNext Template

Also note that we have ASP.NET vNext Class Library as well as ASP.NET vNext Console Application. We will check out these templates shortly.

NEW MODIFIED FILES IN THE SOLUTION EXPLORER

Once the project is created, let's look at the Solution Explorer and observe the project structure.

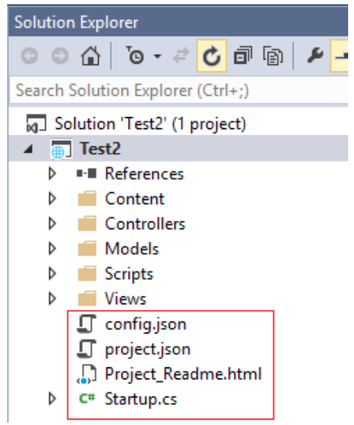


Figure 2: Visual Studio 2014 Solution Explorer

Notice the change . It does not have any web.config or Global.asax files. Instead it has config.json, project.json and startup.cs files which are new in our Project Structure. We will look at each file one by one.

STARTUP.CS

Open Startup.cs file and observe the following code –

```
public class Startup
{
  public void Configure(IBuilder app)
  {
    // Enable Browser Link support
    app.UseBrowserLink();

    // Setup configuration sources
    var configuration = new Configuration();
    configuration.AddJsonFile("config.json");
    configuration.AddEnvironmentVariables();

    // Set up application services
    app.UseServices(services =>
    {
      // Add EF services to the services container
      services.AddEntityFramework()
      .AddSqlServer();

      // Configure DbContext
```

```
services.SetupOptions<DbContextOptions>(options
=>{
  options.UseSqlServer(
  configuration.Get
  (“Data:DefaultConnection:ConnectionString”));
  });

  // Add Identity services to the services
  container
  services.AddIdentity<ApplicationUser>()
  .AddEntityFramework<ApplicationUser,
  ApplicationDbContext>()
  .AddHttpSignIn();

  // Add MVC services to the services container
      services.AddMvc();
});

// Add static files to the request pipeline
app.UseStaticFiles();

// Add cookie-based authentication to the request
pipeline
    app.UseCookieAuthentication(new
    CookieAuthenticationOptions
    {
      AuthenticationType =
      DefaultAuthenticationTypes.ApplicationCookie,
      LoginPath = new PathString(“/Account/Login”),
    });

    // Add MVC to the request pipeline
    app.UseMvc(routes =>
    {
      routes.MapRoute(
        name: “default”,
        template: “{controller}/{action}/{id?}”,
        defaults: new { controller = “Home”, action =
        “Index” });

      routes.MapRoute(
        name: “api”,
        template: “{controller}/{id?}”);
    });
  }
}
```

The startup class contains a *Configure()* function. You can make use of this function to configure the HTTP pipeline. The code also enables Browser Link support. Browser Link has been explained here http://www.dotnetcurry.com/showarticle.aspx?ID=916 but in short, this feature uses SignalR under the hood to create an active connection between Visual Studio and the Browser(s) through which, Visual Studio can update Browsers when any markup change is affected.

Configuration settings are fetched from config.json file. We will take a look at config.json file shortly. You can configure various services that your application needs during development. For example, by default Entity Framework is added as a service and is configured to persist the data into SQL Server. In case you are testing your application and you don't want SQL Server, you can change that to in-memory storage as well. The code shown in Figure 3 does that:

```
// Set up application services
app.UseServices(services =>
{
    // Add EF services to the services container
    services.AddEntityFramework()
    .AddInMemoryStore();
        //.AddSqlServer();
```

Figure 3: Add application services

Next is adding the MVC Service, Identity Service and MVC routes in our configure function. Similarly whatever services your application requires, can be added to this Configure function.

PROJECT.JSON

project.json lists the dependencies for the application like Nuget packages or framework references and some optional configuration. Open the References folder in your project shown in Figure 4.

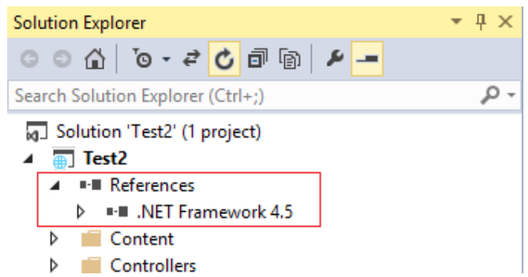


Figure 4: Project References

Initially you will see only .NET Framework 4.5. Drill down by clicking the arrow and you will see more references on which your project is dependent. Actually these are all NuGet packages.

All these dependencies are listed in project.json.

```
{
  “dependencies”: {
    “Helios”: “0.1-alpha-build-0585”,
    “Microsoft.AspNet.Mvc”: “0.1-alpha-build-1268”,
    “Microsoft.AspNet.Identity.Entity”: “0.1-alpha-
    build-1059”,
    “Microsoft.AspNet.Identity.Security”: “0.1-alpha-
    build-1059”,
    “Microsoft.AspNet.Security.Cookies”: “0.1-alpha-
    build-0506”,
    “Microsoft.AspNet.Server.WebListener”: “0.1-alpha-
    build-0520”,
    “Microsoft.AspNet.StaticFiles”: “0.1-alpha-
    build-0443”,
    “Microsoft.Data.Entity”: “0.1-alpha-build-0863”,
    “Microsoft.Data.Entity.SqlServer”: “0.1-alpha-
    build-0863”,
    “Microsoft.Framework.ConfigurationModel.Json”:
    “0.1-alpha-build-0233”,
    “Microsoft.VisualStudio.Web.BrowserLink.Loader”:
    “14.0-alpha”
  },
  “commands”: {
    /* Change the port number when you are self hosting
    this application */
    “web”: “Microsoft.AspNet.Hosting --server
    Microsoft.AspNet.Server.WebListener --server.urls
    http://localhost:5000”
  },
  “configurations”: {
    “net45”: {
      “dependencies”: {
        “System.Data”: “”,
          “System.ComponentModel.DataAnnotations”: “”
      }
    },
    “k10”: {
    }
  }
}
```

Here's how the dependencies are mapped, as shown in Figure 5.

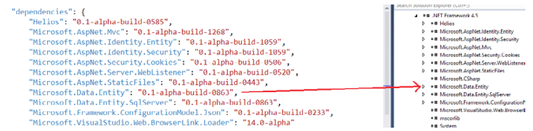


Figure 5: Remove Dependencies

Let's make a small change to this .json file and then we will observe the effect of our change. Open the json file and under dependencies section, remove the “Microsoft.Data.Entity” entry. You will observe that when you remove this entry, it automatically removes the references from the Solution Explorer too as seen in Figure 6.

Figure 6 : Dependency mapping

Now add an Entity Framework entry with the latest version by specifying 0.1-alpha-build-* as shown Figure 7.

```
"dependencies": {
    "Helios": "0.1-alpha-build-0585",
    "Microsoft.AspNet.Mvc": "0.1-alpha-build-1268",
    "Microsoft.AspNet.Identity.Entity": "0.1-alpha-build-1059",
    "Microsoft.AspNet.Identity.Security": "0.1-alpha-build-1059",
    "Microsoft.AspNet.Security.Cookies": "0.1-alpha-build-0506",
    "Microsoft.AspNet.Server.WebListener": "0.1-alpha-build-0520",
    "Microsoft.AspNet.StaticFiles": "0.1-alpha-build-0443",
    "Microsoft.Data.Entity": "0.1-alpha-build-*",
    "Microsoft.Data.Entity.SqlServer": "0.1-alpha-build-0863",
    "Microsoft.Framework.ConfigurationModel.Json": "0.1-alpha-build-0233",
    "Microsoft.VisualStudio.Web.BrowserLink.Loader": "14.0-alpha"
```

Figure 7: Adding entry in project.json

Observe the activity of restoring the packages in the *Output* window. Whenever you add or remove dependencies, Visual Studio 2014 will automatically add or removes the packages.

In the project.json file, you can see there are two other sections; commands and configurations. *Commands* is a list of commands which can be invoked from command line, for example for self-hosting. *Configuration* section allows to you build the project against –

- net45 – Presents the full .NET Desktop CLR.
- k10 – Presents the cloud optimized CLR.

The biggest advantage of the project.json is that now it makes

it easier to open vNext projects in an editor of your choice or even in the cloud.

CONFIG.JSON

The config.json file contains the application and server configurations which was earlier found in the web.config file.

```
{
  "Data": {
    "DefaultConnection": {
      "ConnectionString": "Server=(localdb)\\
      mssqllocaldb;Database=aspnetvnext-
      a2861727-a7e4-4543-bc22-a409386e6f92;
      Trusted_Connection=True;
      MultipleActiveResultSets=true"
    }
  }
}
```

.KPROJ

When you create a project in Visual Studio, a .kproj file will be placed in the root of the project. This file is used to persist settings which are needed by Visual Studio to load/run your application.

DYNAMIC CODE COMPILATION [ROSLYN COMPILER]

There are two types of compilation used by ASP.NET Applications – Project compilation (.csproj/.vbproj) and Runtime Compilation (when app is running). Starting with vNext, we now have the Roslyn compiler which will potentially improve the application startup/pre-compilation time. Due to dynamic compilation available in Roslyn, you don't need to recompile your apps to see the changes you have made in the browser. Up till now whenever we changed code inside Visual Studio, we would hit Ctrl+Shift+B and then refresh the page. However with Roslyn, now you just make the changes, hit F5 in your browser and your changes are reflected, *without* rebuilding the project.

Let's see this with an example. Run your web site (Ctrl + F5). Now open the Home controller in Visual Studio and change some code. The code with changes is shown here –

```
public IActionResult Index()
{
  return Content("<h1>Hello DNC Mag Viewers!!");
}
```

Save the application and just refresh the page. You will see the changes as shown here:

Those familiar with Node.js programming will be delighted to see this no-compile feature!

ASP.NET VNEXT CLASS LIBRARY

Now this experience is not just limited to web projects. You will get a similar experience in a class library as well. In our demo, we will add a class library and then add a reference to the same in our project. To add a Class library, right click the Solution in Solution Explorer and add a new project. Choose ASP.NET vNext Class library as shown in Figure 8.

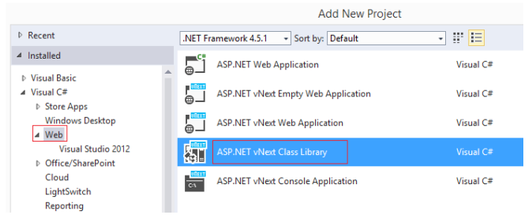


Figure 8:Add vNext Class Library

The default class code is as shown here –

```
namespace Classlibrary1
{
  public class Class1
  {
    public Class1()
    {

    }
  }
}
```

Look at the project structure in your Solution Explorer and locate the project.json file. We will need to reference the library in our project. If you right click References and try adding the library, you will see the following context menu as shown in Figure 9.

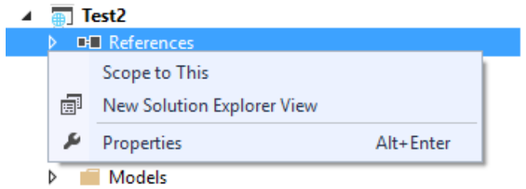


Figure 9: Reference Context Menu

A menu item to add the library reference does not exist! Then how do you add the reference? You guessed it right. Open project.json and add the reference as a dependency as shown in Figure 10.

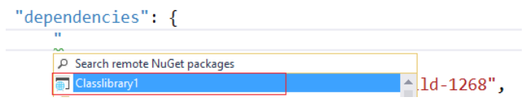


Figure 10: Reference Library as dependency in project.json

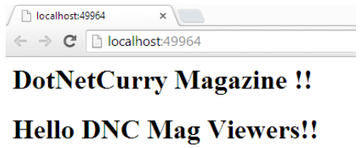
You will find the ClassLibrary1 reference under .NET Framework 4.5. We will go back to Class library and add a method as shown here –

```
public static string Message()
{
  return "<h1>DotNetCurry Magazine !!</h1>";
}
```

In our Home controller, we will call this method. Change the Index method to the following –

```
public IActionResult Index()
{
  return Content(Classlibrary1.Class1.Message() +
  "<h1>Hello DNC Mag Viewers!!");
}
```

Press Ctrl+F5 to run the application and see the output:



Now we will change the method in the class library and save it. However we will not press Ctrl+Shift+B. Just refresh the application and the changes will appear in our output. The code is as shown below –

```
public static string Message()
{
  return "<h1>Anniversary DotNetCurry Magazine !!</
  h1>";
}
```

So the same dynamic compilation feature is available here too. Now take a look at the Bin/Debug folder of the class library.

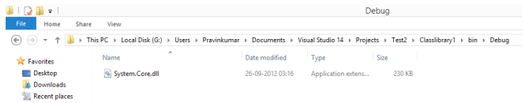


Figure 11: Bin/Debug folder of the class library

If you see, there is no .DLL or .PDB file. This is because the compilation is dynamic. To see this in action, we will put a break point into our class library function and check if the break point hits or not.

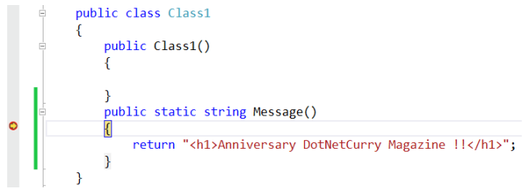


Figure 12: Debugging vNext Class Library

How did this break-point hit when we have no debug file (.pdb)? Let's go to Debug > Windows > Modules menu. You will see all other libraries with our class library as shown Figure 13 with the symbol and its status.

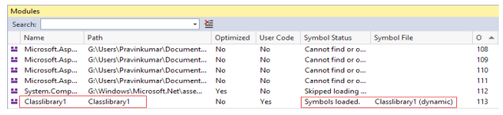


Figure 13 – Modules Window

SELF-HOSTING ASP.NET VNEXT APPLICATION

Until now, Web API 2 and SignalR 2 already supported self-hosting. In .NET vNext, since your app contains the framework and the libraries needed for the app to run, you can now self-host even ASP.NET apps. Let's take a look at how to run the ASP.NET vNext App from a command prompt. I have already configured the command line tool on my machine. You can do the same by following the instructions from https://github.com/aspnet/Home

Open the Command Prompt. The first command which I am running is – kvm list which shows you the available .NET Frameworks on your machine. On my machine, I have two of them –

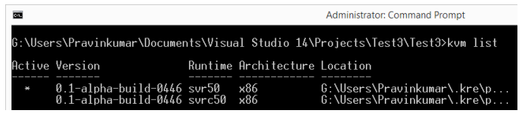


Figure 14: kvm list

In Figure 14, the active column (with the asterix *) shows the framework I am currently using. You can change it to *default* as shown in Figure 15

Figure 15: Default kvm

You can run ASP.NET vNext applications using the *k run* or *k web* commands. Let's start an application using the command 'k web' as shown in Figure 16.

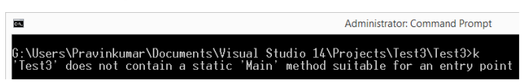


Figure 16: Starting an application

The web command is defined in the project.json file:

```
"commands": {
  /* Change the port number when you are self hosting
  this application */
  "web": "Microsoft.AspNet.Hosting --server Microsoft.
  AspNet.Server.WebListener --server.urls http://
  localhost:5000"
}
```

The *k web* command starts the HTTP listener without the need to explicit build the application, by compiling the code on the fly. This command actually tells to look for a *main* in Microsoft.AspNet.Hosting API and run the host on the host URL. In our case, it is running on port number 5000.

Now open the browser and type http://localhost:5000 and see the magic. Your application is now running. If you look at the status bar icons, you will not see IIS Express running. Awesome!

PUBLISHING ASP.NET VNEXT APPLICATION

We can now take our entire application and move it to some other machine, just like that. Let's see how. Right click on the application and click on Publish as shown in Figure 17.

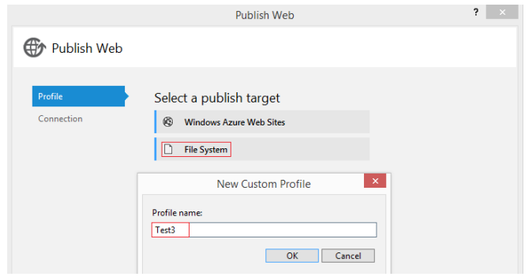


Figure 17: Publish Wizard

Specify the path and click on the OK button. Look at the publish details and you will see that the runtime also gets copied, which you can take and run on any machine. The activity details recorded are as shown in Figure 18.

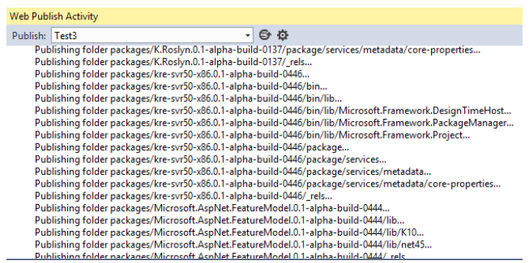


Figure 18: Web Publish Activity details

Now let's observe the deployment folder. Open the C:\AppCopy and you should see the batch file. This is the file which contains the command from project.json. Now open the packages folder and you will see all the packages with the runtime as shown in Figure 19.

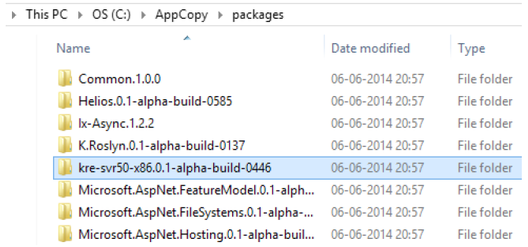


Figure 19: Deployed output folder with packages and runtime

ASP.NET MVC 6.0 NEW FEATURES AND CHANGES

In ASP.NET vNext; MVC, Web API, and Web Pages frameworks will be merged into one framework, called MVC 6. Let's quickly see some new features of ASP.NET MVC 6 as described at http://www.asp.net/vnext/overview/aspnet-vnext/overview .

Open the Startup.cs file. The code below enables ASP.NET MVC.

```
public void Configure(IBuilder app)
{
  app.UseServices(services =>
  {
    // Add MVC services to the services container
        services.AddMvc();
  });
  // Add MVC to the request pipeline
  app.UseMvc(routes =>
  {
    routes.MapRoute(
      name: "default",
      template: "{controller}/{action}/{id?}",
      defaults: new { controller = "Home", action =
      "Index" });

    routes.MapRoute(
      name: "api",
      template: "{controller}/{id?}");
  });
}
```

If you comment out services.AddMvc() method and run the application, you will see the following output as shown in Figure 18.

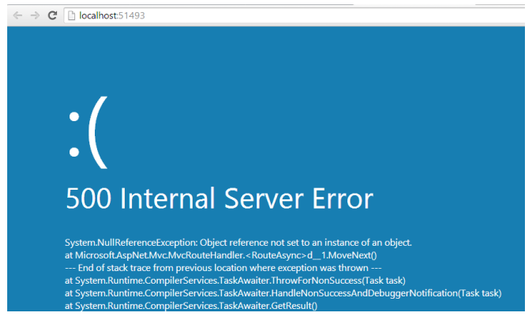


Figure 20: Internal Server Error in MvcRouteHandler

By the way, observe the new error page!
Now let's observe the route configuration. In previous MVC versions, we had the following route configuration –

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id =
      UrlParameter.Optional }
);
```

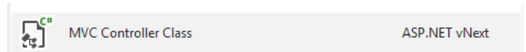
Figure 21: Route configuration in versions before MVC 6

In ASP.NET MVC 6, the route configuration gets changed to the following:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller}/{action}/{id?}",
        defaults: new { controller = "Home", action = "Index" });

    routes.MapRoute(
        name: "api",
        template: "{controller}/{id?}");
});
```

Figure 22: ASP.NET MVC 6 Route Configuration

Instead of UrlParameter.Optional, in ASP.NET MVC 6, we now have a question mark **?** which means *id* is optional.
Now add a new controller with the name Customers and create a view for the same.

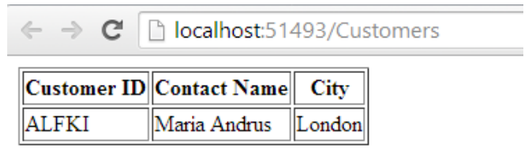


When you add a controller, you will see only one action method with the name "Index" as shown here:

```
public class CustomersController : Controller
{
  // GET: /<controller>/
  public IActionResult Index()
  {
    return View();
  }
}
```

In order to add a view, we will add a new folder with the name Customers into our Views folder and add Index.cshtml file in the folder. Write the following code:

```
@{
    var customers = new
    {CustomerID=”ALFKI”,ContactName=”Maria
    Andrus”,City=”London” };
}

<!DOCTYPE html>

<html lang=”en”>
<head>
  <meta charset=”utf-8” />
  <title>Index Page</title>
</head>
<body>
  <table border=”1”>
    <tr>
      <th>
        Customer ID
      </th>
      <th>
        Contact Name
      </th>
      <th>
        City
      </th>
    </tr>
    <tr>
      <td>@customers.CustomerID</td>
      <td>@customers.ContactName</td>
      <td>@customers.City</td>
    </tr>
  </table>
</body
</html>
```



You can also add a model to the app and pass it to the view. Let’s add a Customers class and pass it to the view:

```
public class Customers
{
    public string CustomerID { get; set; }
    public string ContactName { get; set; }
    public string City { get; set; }
}
```

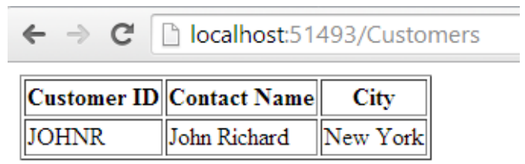
Now change the Index method by passing an instance of Customers to the View method:

```
public IActionResult Index()
{
  Customers customer = new Customers()
  {CustomerID=”JOHNR”, ContactName=”John
  Richard”,City=”New York” };
  return View(customer);
}
```

The final step is using this model in our View. Change the view to the following:

```
<!DOCTYPE html>

<html lang=”en”>
<head>
  <meta charset=”utf-8” />
  <title>Index Page</title>
</head>
<body>
  <table border=”1”>
    <tr>
      <th>
        Customer ID
      </th>
      <th>
        Contact Name
      </th>
      <th>
        City
      </th>
    </tr>
    <tr>
      <td>@Model.CustomerID</td>
      <td>@Model.ContactName</td>
      <td>@Model.City</td>
    </tr>
  </table>
</body>
</html>
```

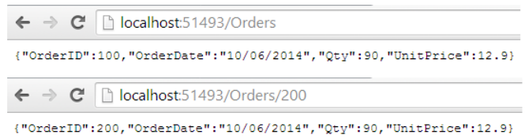
The output is as shown here:



If the route template does not include {action}, the framework uses HTTP verb to select the action name. This is very similar to the style of routing in ASP.NET WebAPI 2. Read more about it here http://www.dotnetcurry.com/ShowArticle.aspx?ID=989 I have created a controller with the name Orders and the controller code is as shown here:

```
public class OrdersController : Controller
{
  // GET: /<controller>/
  public IActionResult Get()
  {
    var orders = new { OrderID = 100, OrderDate =
    “10/06/2014”, Qty = 90, UnitPrice = 12.90 };
    return Json(orders);
  }

  public IActionResult Get(int id)
  {
    var orders = new { OrderID = 200, OrderDate =
    “10/06/2014”, Qty = 90, UnitPrice = 12.90 };
    return Json(orders);
  }
}
```

The output is the following:



In ASP.NET MVC 6, the Controller need not be derived from the Microsoft.AspNet.Mvc.Controller base class. You can write a plain CLR class or POCO as shown here:

```
public class SampleController
{
  public SampleController(IActionResultHelper helper)
  {
    MvcHelper = helper;
  }
  private IActionResultHelper MvcHelper { get; set; }
  public ActionResult Index()
  {
    return MvcHelper.Content(“<h1>WelCome To
    DotNetCurry Magazine</h1>”, null, null);
  }
}
```

The output is shown here:

In the code we just saw, we are injecting the IActionResultHelper interface, which is a helper for creating action results, via a constructor.

In earlier versions of ASP.NET MVC, we have had a way to manage large MVC projects, using *Areas*. For example, if we are working with a Purchase Order System, we can divide large parts of the application, like Shipping, Placing Orders and Generating Invoice etc into sections. These sections can be divided into number of areas. These areas are placed into a special folder called Areas and are registered by calling an AriaRegistration.RegisterAllAreas method into Global.asax file and Application_Start event. In MVC 6, we can now decorate the controllers with [Area] attribute and create a route template using {area} parameter. Let’s look at an example here –

```
[Area(“POSupply”)]
public class SuppliersController : Controller
{
```

```
  public ActionResult Index()
  {
    return View();
  }
}

Route template should look like the following –
app.UseMvc(routes =>
{
  routes.MapRoute(
    name: “OPAreas”,
    template: “{area}/{controller}/{action}”);
}
```

CONCLUSION:

So that was a quick overview of what's new in ASP.NET vNext using Visual Studio 2014 CTP. We covered quite a lot of ground. Here's a quick recap:

- Introduction to Web Templates in Visual Studio 2014 CTP
  - ASP.NET vNext Web Application
  - ASP.NET vNext Class Library
- Introduction to Project Structure
  - Startup.cs
  - project.json
  - config.json
- Dynamic Code Compilation
- Self-hosting vNext App
- Publishing vNext Application
- ASP.NET MVC 6 New Features and Changes –
  - Enabling ASP.NET MVC
  - Routing optional parameter with question mark (?)
  - New Error Page
  - Introduction to Controller, View and Model
  - Exposing JSON
  - [Area] attribute
  - Controller with IActionResultHelper interface

These are still early days, so expect bugs and broken functionality. It's a developers delight to get these early previews to work on, so make sure you explore and test-drive what's going to be the next generation of .NET! ■

Pravinkumar works as a freelance corporate trainer and consultant on Microsoft Technologies. He is also passionate about technologies like SharePoint, ASP.NET, WCF. Pravin enjoys reading and writing technical articles. You can follow Pravinkumar on Twitter @pravindotnet and read his articles at bit.ly/pravindnc