ISSUE 14 | SEP-OCT 2014 EDITION

DNCMagazine

www.dotnetcurry.com

Real-time Data Visualization using D3 and ASP.NET SignalR

C# vNext

Continuous Integration Continuous Deployment with TFS 2013

USING REST in SHAREPOINT APP to perform CRUD Operations

XAMARIN.FORMS

SOFTWARE GARDENING: INSECTICIDE

BACKBONE.JS at a glance

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

GROUPDOCS.TOTAL

Professional APIs that allow developers to empower their apps with document collaboration capabilities.

Your Document Collaboration APIs

GroupDocs.Viewer

Native-text, high-fidelity HTML5 document viewer with support for over 49 file formats.

GroupDocs.Signature

Electronic signature API that gives your apps legally binding e-signature capabilities.

GroupDocs.Conversion

Universal document converter for fast conversion between more than 49 file formats.

GroupDocs.Annotation

A powerful API that lets developers annotate Microsoft Office, PDF and other documents within their own apps.

GroupDocs.Assembly

Incorporates data entered by users through online forms Into both Microsoft Office and PDF documents.

GroupDocs.Comparison

A diff view API that allows end users to quickly find differences between two revisions of a document.

100% Standalone - No Office Automation

Download Now

.NET Libraries | Java Libraries | Cloud APIs | Cloud Apps

SALES INQUIRIES: +1 214 329 9760

sales@groupdocs.com

GROUPDOCS
Your Document Collaboration APIs

www.groupdocs.com

CONTENTS

Suprotim Agarwal

Editor in Chief

One of the most important aspects of the modern web is it's openness, which has inspired and enabled a fusion of divergent technologies and opposed proprietary solutions. Companies like Microsoft and Adobe have embraced and chosen the open web over it's proprietary technologies like Silverlight and Flash. These companies are on a mission to shape the future of the web by engaging and participating with the web standards groups, open source projects and browser vendors.

One such attempt to evolve the web has been made in the form of a web platform called HTML5. To strengthen and support this new platform, complimentary technologies like CSS, JavaScript and SVG are being used and evolved. To reap the advantage of a server-side language like MVC, you must compliment it with client-side technologies. With the modern client browsers being extremely feature rich and capable of doing a great deal of processing for your application, you will reduce the load on your servers and optimize CPU utilization as well as save bandwidth costs.

We are in a time of unprecedented innovations on the web and HTML5 and JavaScript are two frontrunners in this innovation. As a .NET Developer, it is of utmost importance to understand and apply these new standards and technologies, as it will prepare you for the future of web development. Don't worry if you haven't invested much time in these technologies, as you haven't missed the bus, not yet.

Editor In Chief Suprotim Agarwal
suprotimagarwal@dotnetcurry.com

Art Director Minal Agarwal
minalagarwal@a2zknowledgevisuals.com

Contributing Writers Craig Berntson, Filip Ekberg, Gil Fink, Mahesh Sabnis, Nish Anil, Ravi Kiran, Subodh Sohoni

Reviewers: Alfredo Bardem, Craig Berntson, Richard Banks, Suprotim Agarwal, Todd Crenshaw

Next Edition 1st Nov 2014
www.dncmagazine.com

C# vNext

CODENAME "ROSLYN"

For many years now, Microsoft has been working on a re-write of their C# and VB compiler. We have known about it for some years now and it came as a big surprise recently when Microsoft announced that they would open source their new compilers. Not only has this meant that the work they are doing is transparent, it also means that the community can give feedback on what they see fit. As the language design documents are available on the CodePlex site, any developer wanting to dig into the decisions made when adapting new features, can do so. With the rewrite of the C# and VB compiler, it has become so much easier for the language design team to add new language features. Given that the cost of adding a new feature is substantially smaller than what it was with the old compilers, the language design team can focus on making life much easier for us developers and it does not necessarily mean changing anything from the runtimes perspective.

Up until now, the C# and VB compilers have really been a black box to the consumers. This black box has translated a given syntax into something that is executable or something that we can simply reference. In a way, this has really limited the interoperability of the compiler, which is OK for most people. Although, you will be amazed when thinking about the opportunities that opens up when having the compiler completely open and accessible to the consumer. Before we look at what the next version of C# will look like, let us think around what these changes will do to us; the consumers of the compiler.

Microsoft has given the new compilers the codename **"Roslyn"**, a name that really has a nice ring to it. Imagine being much free in the ways we can interop with compiler, the possibilities are endless as we can "hook in" to the compiler and modify certain behavior as our programs compile. These modifications to the compiler's behavior are known as analysis and refactoring plugins. When opting in and installing Roslyn, you get sample code and solutions. One of these samples lets you detect if your solution contains declarations that could be converted to constants. Another example will let you detect variable names containing certain characters; this is to educate you in how the compiler APIs work.

Whilst this article is not about how we write these kinds of plugins, it is important to understand the impacts of having a re-written compiler for C# and VB. As it makes it easier for consumers to interop with the compilation and analysis process, it makes it easier for Microsoft to introduce language features and fix possible bugs in the language. Hence, this makes it easier for Microsoft to focus on giving the C# and VB developers the best experience possible when using their languages.

These changes are what we will spend the rest of this article discussing, some of which may not be coming in vNext at all, some of them are already implemented and ready to go. As Roslyn is open source and available on CodePlex, Microsoft invites the community to supply feedback on their design decisions and they are truly listening to the C# and VB community.

WHAT IS NEW IN C# VNEXT?

We will not necessarily see new keywords in C# vNext, but we will see semantic differences that will make it easier for developers to avoid boilerplate code. Some of these introduced semantic differences are much awaited and some of them are solving thinner edge cases; they are all very welcomed by the community! As the next version of the compiler is still under development, some of the language features may be taken out, some may be added and some may be delayed to the version after vNext. The evolution of C# is a proof that the language lives, people are using it and it is here to stay.

Primary constructors

Far too often, you will find yourself creating constructors that are simply delivering values to private fields to be used later on in the class. Consider that we have a class representing a Person; a person is required to have a name and an age. This might of course not be the case in reality where we would probably require more information to represent a person, but in this case, this will suffice.

The constructor for the person can be written in a straightforward manner. We are not requiring any particular validation on the data passed to the constructor at this point, as seen in the following code sample.

```
public Person(string name, int age) {
  _name = name;
  _age = age;
}
```

Would it not be great if we did not have to specify this constructor as it is really the default, minimal requirement for our class? Sure enough, it would and there is now a much easier way to define our classes and their primary constructor. It requires us to step out of the scope of the constructor and change the signature of the class. We are now diverting a bit from what we are used to C# looking like.

We need to change the signature of the class to include the parameters we would like to have in our primary constructor, in this case we have the name and the age. The following is what our class will now look like.

```
class Person(string name, int age){
  private string _name = name;
  private int _age = age;
}
```

The parameters we define on the primary constructor sits on the class declaration, we need to assign these to instance variables. After this, we can use our instance variables _name and _age like any normal instance variable.

In case you want to have more than one constructor, you define the most used (hence primary) on the class declaration and then define the other constructors as we normally do. See the following code sample.

```
class Person(string name, int age) {
  private string _name = name;
  private int _age = age;
  private string _address;
  public Person(string name, int age, string address) :
  this(name, age) {
    _address = address;
  }

  public void Speak() {
    Console.WriteLine("Hi, my name is {0} and I am {1}
    years old", _name, _age);
  }
}
```

Interesting enough, if we open up the compiled assembly in reflector to inspect what code was generated by the compiler, we will see that it is just normal code as we have seen before. This means that there in fact does not need to be a difference in the runtime!

```
internal class Person{
  private string _address;
  private int _age;
  private string _name;
```

```
  public Person(string name, int age)  {
    this._name = name;
    this._age = age;
  }

  public Person(string name, int age, string address) :
  this(name, age)  {
    this._address = address;
  }

  public void Speak()  {
    Console.WriteLine("Hi, my name is {0} and I am {1}
    years old", this._name, this._age);
  }
}
```

Auto-properties

Properties are so commonly used in C# that you may think there is no optimization left to be done. However, with the introduction of primary constructors we need a way to initialize the properties that we might want to do on instantiation of our class. This is can be done with *auto-properties* initializers.

As seen in the following code sample, we simply say that the property equals the value we give it.

```
class Person(string name, int age) {
  public string Name { get; set; } = name;
}
```

This also works for getters only, as seen in the following code sample.

```
class Person(string name, int age) {
  public string Name { get; } = name;
}
```

We know that properties are backed by fields, and the same will happen here but the compiler will handle it for us. Looking at the generated code by the compiler, we will see that it moves the initialization into the primary constructor and sets the properties backing field to the correct value.

```
private readonly string <Name>k__BackingField;
public string Name {
  get  {
    return this.<Name> k__BackingField;
  }
}

public Person(string name, int age) {
  this.<Name> k__BackingField = name;
  this._age = age;
}
```

The auto-property initializers, together with the primary constructors, lets us avoid some boilerplate code. It is not a lot, but it will make a huge difference.

Using statement for static members

If you are using a particular static member a lot, you will find yourself repeating the class name or the path to that static member repeatedly. It gets a bit redundant and it would certainly be great if we could include classes static members as a part of our own type, making it easier to access the methods. Consider the usage of `Debug.WriteLine` or `Console.WriteLine` for instance. Would it not be perfect if we could define that we are including all static members of Debug and Console, making it usable inside our class, without the full path to the static member?

As seen in the following code snippet, this is something that we certainly can do with C# vNext. We simply define that we want to use `System.Console` and we immediately just have to write `WriteLine` instead of the full path.

```
using System.Console;

class Person(string name, int age) {
  public string Name { get; } = name;
  private int _age = age;

  public void Speak()  {
    WriteLine("Hi, my name is {0} and I am {1} years
    old", Name, _age);
  }
}
```

Again, this change is not as major as introducing `async` and `await`, but it will remove the boilerplate code that we always have to write. Another major benefit this gives us is that we can much more easily swap which `WriteLine` is being used, `Debug.`

`WriteLine` as opposed to `Console.WriteLine` means we just have to change the *using* statement. Of course, we could also introduce the same method inside our class and that would be used instead of the static method. This means if we introduce a method called `WriteLine` with the same parameter as the one on the static member we previously accessed, the method on our class will be used instead.

You can only do this for static classes, so you cannot for instance do a `using` for `System.DateTime`.

Dictionary initializer

When using dictionaries, sometimes you want to initialize them with values just as you can do with arrays and lists. Previously this has been cumbersome as you have had to call the add method for each item you want to add. It is not really a deal breaker, but there is room for improvement and again the language design team has delivered. We can now initialize our dictionaries using something called *dictionary initializers* which work very similar to array initializers.

You will see in the following code sample that these new initializers even work with instance variables.

```
class Person(string name) {
  private Dictionary<string, string> _data =
  new Dictionary<string, string> {["Name"] = name };
}
```

It will of course also work with the *auto-property initializers* that we saw previously in this article. These new language features work so nicely together that it makes the language even more pleasant to work with!

```
class Person(string name, int age) {
  public Dictionary<string, string> Data { get; }
  = new Dictionary<string, string> {["Name"] = name };
}
```

Declaration expressions

Have you ever encountered the *out* keyword? Then you know that you have to declare it before using it in a method call where there is an *out* parameter. In many cases, personally I would just like to say that I want the variable to be created inline, instead of having to declare it in the line before.

Consider the following code; normally you would have to declare a variable for the result before calling the actual method.

```
public void CalculateAgeBasedOn(int birthYear, out int
age) {
  age = DateTime.Now.Year - birthYear;
}
```

You would end up with something like you can see in this following code snippet, which is still not something that is deal breaking, but it breaks the flow.

```
int age;
CalculateAgeBasedOn(1987, out age);
```

We no longer have to do that, as you would most likely have already figured out. Instead we can say that this variable is created inline in the following manner:

```
CalculateAgeBasedOn(1987, out var age);
```

It now goes down to one line of code instead of two. Looking at the generated code by the compiler though, it still generates the same code as we have in the first example we looked at, where we are introducing the variable before the call. It is just nice that we as developers do not really have to bother doing it.

Using await in a catch or finally block

If you have caught on with *async* and *await* you may have experienced that you want to await for some result in a catch or finally block. Consuming asynchronous APIs are getting more and more common as it is getting much easier with the additions in .NET 4.5. With this being more common, we need a way to ensure that it is compatible with all different scenarios.

Consider logging for instance. You may want to write a file log when you catch an error but not hold up the caller for too long. In this scenario, it would be great to have the ability to await an asynchronous call inside the catch block.

Equally when talking about the finally block, we may want to clean up some resources or do something particular inside the finally block that invokes an asynchronous API. As seen in the

following code sample, this is now allowed.

```
public async Task DownloadAsync() {
  try {}
  catch {
    await Task.Delay(2000);
  }
  finally {
    await Task.Delay(2000);
  }
}
```

Filtering Exceptions with Exception Filters

When catching exceptions, you may want to handle the exceptions differently depending on, for instance, severity in the exceptions. Of course, this has been straightforward before by just having conditional code inside the catch block. Although, would it not be easier if we could filter the exceptions? Indeed, it would, and this has now been introduced as something known as *exception filters*.

As seen in the following code sample, we simply apply the filter after we have said what exception we are catching.

```
try {
  throw new CustomException { Severity = 100 };
}
catch (CustomException ex) if (ex.Severity > 50) {
  Console.WriteLine("*BING BING* WARNING *BING BING*");
}
```

For this example to work, a class called CustomException was introduced with only one property on it called Severity. In this case, we are looking for a severity over 50, what happens when the severity is less than 50? The exception will not be caught. To fix this, we can introduce another catch block after it, just as we have been able to do before when catching multiple different exceptions.

```
try {
  throw new CustomException { Severity = 100 };
}
catch (CustomException ex) if (ex.Severity > 50) {
  Console.WriteLine("*BING BING* WARNING *BING BING*");
}
catch (CustomException ex) {
  Console.WriteLine("Whooops!");
}
```

Keep in mind that the order matters, you want to order them descending where the first one is the most specific one going down to less specific filters as seen in the next code sample.

```
catch (CustomException ex) if (ex.Severity > 50) {}
catch (CustomException ex) if (ex.Severity > 25) {}
catch (CustomException ex) {}
catch (Exception ex) {}
```

PLANNED FEATURES IN C#

Up until now, we have looked at features that have been implemented in the new compiler, features that you can experiment with yourself by either downloading the extension for Visual Studio or installing the CTP of Visual Studio 14. There are more features on the table though, some of the features have been talked about from the beginning of the language design but have later been cancelled, or only introduced in VB instead of C# as C# may already have had it or it would not make any sense to have it in C#, to start with.

The rest of the features that we will look at are either planned, to be implemented or are just on the table and might get into the next version of C#. There is no guarantee that a feature that is planned to be implemented will be done by the time of the next language release. It might even end up being completely cancelled, as there may be no point in implementing it. Keep in mind though that the language design team adds their documents from their design meetings to CodePlex, so that all those developers interested in the future of C# can follow along and see what is in it for the future.

NULL PROPAGATION

While this feature is actually already done, it is not currently available in the extension for Visual Studio available for download. Personally, it has happened so often that I do not think I can count on my fingers how many times I have gotten a null reference exception due to forgetting to do a null check on one of the properties I need to use, especially when it is a long path on a certain object that leads down to the property that I

want to use.

Consider that we have a person, the person has an address and we want to get the street name. The address is represented by an object and if the person has not given us their address, the property for the address on the person is null. Now you would most likely think that this is bad design, surely enough it may be. If you would take up the argument with a domain-driven development evangelist, they would tell you that the person is not a valid object until all the fields have their correct values.

Being stuck in the world of null's, we need a better way of handling these type of scenarios and this is where *null propagation* comes into play. Have a look at the following code sample. It is a representation of what we just talked about; the person with an address.

```
class Person {
  public string Name { get; set; }
  public Address Address { get; set; }
}
class Address {
  public string AddressLine1 { get; set; }
  public string AddressLine2 { get; set; }
}
```

We can now create an instance of a person, without giving it a valid address, as seen in this code sample.

```
var filip = new Person {
  Name = "Filip"
};

Console.WriteLine(filip.Address.AddressLine1);
```

Of course, this will throw a null reference exception. Fixing this previously would mean we had to add inline null checks, which could be pretty messy as seen in the following code sample.

```
Console.WriteLine(filip.Address == null ? "No Address" :
filip.Address.AddressLine1);
```

With null propagation, we simply have to add one character to fix this problem, which is nice on the eyes.

```
Console.WriteLine(filip?.Address.AddressLine1 ?? "No
Address");
```

This will be converted to what we saw previously, thus the entire expression will be returned as null if one of the null propagations detects null. This will be extremely handy at times and hopefully we will see less crashes due to invalid null checks.

BINARY LITERALS AND DIGIT SEPARATORS

Most of us probably do not write binary numbers on a daily basis, however for those of us who do, there is a planned feature to make it even easier to write *binary literals* in our code. If you would like to write the binary representation of eight, you could simply define that as 0b00001000. That looks pretty much like the way we write hexadecimal numbers.

Speaking of hexadecimal numbers, we may also get something called *digit separators*. This will be used to separate digits to make it easier on the eyes. Consider having the hexadecimal representation 0xFF00FAAF. To make this easier to read, it is planned to introduce a separator. The separator will make the hexadecimal representation look a bit different, but much easier to read as you see here 0xFF_00_FA_AF.

EXPRESSION-BODIED MEMBERS

If we are using primary constructors, we may want to introduce a property in our class that calculates something every time based on these values. It could be calculating the area of a rectangle based on the width and height passed to the rectangle primary constructor, or it could be calculating something differently.

Expression-bodied members means that we can have members in our class that have an expression body, and is straightforward. As always when introducing expressions and their bodies, we are using the fat arrow syntax.

As seen in the following example, we have a public member that will evaluate an expression when it is called, that calculates the area of a rectangle.

```
class Rectangle(int width, int height){
  public int Area => width * height;
}
```

EVENT INITIALIZERS

Most of us are using events on a daily basis and it is quite irritating that when we have the power of initializers, we cannot apply the same pattern when it comes to events. This means that we have not been able to initialize the events in our normal object initializers.

It is one of the smaller changes, but it makes all the difference in the world. As seen in the following code sample, we have a web client that informs us when a download is completed. We could do this using an object initializer and initialize the event directly in the object initialization.

```
var client = new WebClient {
  DownloadFileCompleted += DownloadFileCompletedHandler
};
```

NAMEOF OPERATOR

In some cases, you may want to retrieve the name of something in your code, be it a member of your class or a static path. The new operator `NameOf` will let us do this. The result of calling `NameOf(Debug.WriteLine)` would be `WriteLine`. This is neat as it does not force us to put strings in our code to represent the name of a member, thus keeping refactoring capabilities intact!

FIELD TARGETS ON AUTO-PROPERTIES

Properties are so widely used as you may still think there is nothing left to improve, however so far it has been impossible to adjust the declaration of the backing field of a property without manually setting it up yourself. An example of how the field targeting would work can be seen in the following code sample.

```
[field: NonSerialized]
public int Age { get; set; }
```

In this case, we would say that the backing field of Age would not be serialized.

FEATURES THAT MAY SHOW UP

As stated earlier, there are features that may or may not show up in the language with the next release. We will not go in to the features that we are not sure of how they will be implemented, but I will list them here:

- Semicolon operator
- Using params with IEnumerable
- Constructor inference
- String interpolation

Since we are not sure about the implementation of these features, it would be pointless to get your hopes up on the above features. They would be much welcomed into the language, but it is certainly not a deal-breaker if they are left out.

Maybe it is something for the version *after* C# 6.

CONCLUSION

If you are ready to try these new language features, you can download the CTP version of Visual Studio 14. The RTM will most likely be released in 2015. There are now a great number of features already implemented; having the compiler written in C# and VB is great for maintainability and extensibility. As seen in the examples given, the language is very much alive and evolving all the time. I personally cannot wait for the next version of C# to come out and be the norm in my work day ■

AUTHOR PROFILE

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

.NET Tools for the Professional Developer

Data Visualization, User Interface, Reporting and Spreadsheet Controls Experts

ActiveReports 8

Create sophisticated, fast and powerful reports

Studio Enterprise

Hundreds of UI controls for all .NET platforms including grids, charts, reports and schedulers

Spread Studio

Flexible and familiar spreadsheet architecture, advanced charting and powerful formula library

Download your free trials at
ComponentOne.com

EXPECT MORE. GET MORE.

ComponentOne®
a division of GrapeCity®

INTRODUCTION

The web development world is changing. Standards such as HTML5, ECMAScript and CSS3 are helping to design more responsive and interactive web applications. Graphics, multimedia and device interactions are first class citizens in web applications and you don't have to use plugins to create sophisticated features. These changes help web developers to build next generation rich internet applications (RIAs).

In the past few years, we have also seen an emerging trend to build Single Page Applications (SPAs). SPAs are web applications that include only one HTML page that acts as a shell to the whole application. This shell is changed via JavaScript interactions made by the users, client-side template engines and client-side routing mechanisms. That means you can create RIAs using HTML, CSS and JavaScript and gain a lot of smart client qualities, such as being stateful and responsive.

One recommended way to create SPAs is by using libraries or frameworks that expose features such as routing, template engines and ways to communicate with the server. The popular SPA frameworks are AngularJS and Ember. One of the popular libraries that help to create SPAs is Backbone.js. In this article we will take a look at Backbone.js and how you can use some of its features. While we won't cover the ways Backbone.js can help you create SPAs, you can find more content about it in "Pro Single Page Application Development" (Apress) book or by searching online.

BACKBONE.JS
AT A GLANCE

WHAT IS BACKBONE.JS?

Backbone.js is a Model-View-Whatever (MVW) library that was written by Jeremy Ashkenas, the creator of CoffeeScript and Underscore.js. Backbone.js is a lightweight JavaScript library that enforces developers to organize their code in a clean and efficient manner and eases the development of SPAs. In its core, Backbone.js includes five main objects that the developer can extend with his/her own functionality:

1. Models – data structure wrappers
2. Collections – sets of models
3. Views – the code behind a HTML view
4. Routers – expose client-side routing mechanism
5. Events – enables to extend any object with event handlers

Each of the objects comes with its own APIs which enables many common development tasks. You can learn more about each of the objects in Backbone.js website: http://backbonejs.org.

Backbone.js also includes an API called sync API that is exposed by Backbone.sync function. This API enables communication with the server, and the model and collection objects use it underneath to synchronize their data when you use their create/update/delete APIs. The sync API is out of scope for this article. Now that you know what Backbone.js offers, we can take a dive into Backbone components.

SETTING UP THE ENVIRONMENT

Before you start working with Backbone.js, you should download its latest version and also download Underscore.js and jQuery. The reason that you will need Underscore.js is that Backbone.js has a dependency on the library and it uses many of Underscore.js utility functions underneath. Backbone.js also depends on jQuery when you extend its View object or when you use its *Router* object; so don't forget to download it as well.

You can download the libraries using the following links:

- Backbone.js: http://backbonejs.org

- Underscore.js: http://underscorejs.org/ (download version should be >= 1.5.0)

- jQuery: http://jquery.com/ (recommended to use version >= 1.9.0)

Now that you have downloaded the libraries, you can create a small HTML file and add a script reference to all the files. The following code example is a template for that HTML web page:

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset=”UTF-8”>
    <title>Webpage Title</title>
</head>
<body>
   <!-- Your html goes here -->
   <script src=”jquery.min.js”></script>
   <script src=”underscore.min.js”></script>
   <script src=”backbone.min.js”></script>
   <script>
     // Your code goes here
   </script>
</body>
</html>
```

Once you have all the libraries in place, let's start by explaining the first Backbone.js object – the *Model*.

THE MODEL

The Model in Backbone.js is just a data structure that acts as a model in the application. The Model responsibility is to store data and expose functionality to manipulate it. If you want to create a Model object, you will first want to create its definition. The following code example shows how to define a Model called *Customer* using Backbone.js API:

```
var Customer = Backbone.Model.extend({
    defaults: {
        name: “”,
        type: “Regular”
    }
});
```

As you can see in the code example, you just need to extend the Backbone.js *Model* object. In the example, you can see that if a Model is initialized without attributes, it will get the defaults set by the option object that was passed to the *Backbone.Model.extend* function.

There is a variety of useful model APIs such as getters, setters, events or the *initialize* function. The following code example will show you how to use some of those API functions:

```
var Customer = Backbone.Model.extend({
  defaults: {
    name: “”,
    type: “Regular”
  },
  initialize: function(){
    console.log(‘model is initialized’); // This
    sentence is printed to the console every time the
    model is initialized
  }
});

var customer = new Customer(); // initialize function
is called here

console.log(customer.get(‘type’)); // no type is
written to the console

customer.set(‘type’, ‘Platinum’);
console.log(customer.get(‘type’)); // Platinum is
written to the console
```

The code is really self-explanatory and very simple to understand by reading the comments added to the code. Now that you know what the Model object is, it is time to learn about another Backbone.js object – the *Collection*.

THE COLLECTION

In Backbone.js, a Collection is a data structure that holds zero to infinite number of Model instances. Like Models, you will define a Collection by extending the *Collection* Backbone.js object. The following code example shows how to create a Collection to the Customer model you saw in the “The Model” section:

```
var Customers = Backbone.Collection.extend({
  model: Customer
});
```

Once you define a Collection, you can start using its instances as sets of Model instances. Collections expose API functionality that you can use to manipulate them. For example, you can add or remove Models using the *add* and *remove* functions or you can retrieve Models by their ‘id’ using the get function. The entire collection API is beyond the scope of this article, but you can always look it up in the Backbone documentation at http://backbonejs.org/#Collection. Collections also expose many of the Underscore.js array utility functions. You can use forEach, sortBy or filter functions to iterate, sort or filter your collections. This is why Backbone.js has a hard dependency on Underscore.js. The following code shows some of these functions in action:

```
var Customers = Backbone.Collection.extend({
  model: Customer
});

// initialize two customer objects
var dave = new Customer({
  name: “Dave”
});

var jane = new Customer({
  name: “Jane”,
  type: “Platinum”
});

// initialize a collection with the previous customer
objects
var customers = new Customers([ dave, jane ]);

// sort the customers by their type and print them to
the console

var sortedByType = customers.sortBy(function (customer)
{
  return customer.get(‘type’);
});

customers.forEach(function(customer) {
  console.log(customer.get(‘type’));
});
```

Now that you are familiar with Backbone.js Models and Collections, we can move on to the next Backbone.js object that we will explore in this article – the *View*.

THE VIEW

The View object in Backbone.js is responsible for displaying the model data inside of an HTML structure. . You can divide your entire web page into small HTML pieces which interact with their own view object and construct a full view. The following code example shows how to define a View object by extending Backbone.js *View* object:

```
var CustomerView = Backbone.View.extend({
  // view configuration
});
```

The view itself uses a template that is rendered by the View logic. In Backbone.js, you use the Underscore.js template engine to create your Views. For example, for a customer view, you might want to use a template such as the following:

```
<script id="customer-template" type="text/template">
  <div>Customer Name: <%= name %></div>
  <div>Customer Type: <%= type %></div>
</script>
```

When this template is rendered, the output will be the customer's name and type, in two different DIVs.

The article doesn't cover the syntax of Underscore.js templates but a good starting point to understand it is the following link: http://underscorejs.org/#template.

In order to compile a template using Underscore.js, you will use the *template* function:

```
var compiledTmpl = _.template($("#customer-template").
html());
```

Once you have a compiled template, you can use it as the template for the View and bind it to a Model.

In a View, there are two properties which are very crucial to understand: *el* and *$el*. The el property is the HTML element that is bound to the View. The *$el* property is a jQuery object that wraps the *el* element and exposes jQuery functions that you can use. The next code example shows how to bind a customer View to an element with a customer id and how to use template engine to compile a template:

```
var CustomerView = Backbone.View.extend({
    el: "#customer",
    template: _.template($("#customer-template").
html())
});
```

In order to render a View, you will need to define a *render* function inside the view options object. The *render* function will use the view element, template and model to create the visualization of the View. The following code example shows how to do that:

```
var CustomerView = Backbone.View.extend({
  el: "#customer",
  template: _.template($("#customer-template").html()),
  render: function () {
    this.$el.html(this.template(this.model.toJSON()));
      return this;
    }
});
```

In the previous example, you can see the use of the model's *toJSON* function to get the model's attributes from the Model. The Model is passed to the compiled template to create binding between both of them. Next, the $el's HTML is set to the output of the binding. One last thing to notice in the *render* function is the return statement that returns the View itself. You should always return the View from the *render* function in order to make it chainable.

Views can also react to events and handle them. Backbone.js View includes an *events* attribute that you can pass in the options objects when you extend the View object. The events attribute is a JavaScript literal object where each of its attribute name indicates the event, and the selector of the element, that triggers the event. The value of the attribute will be the name of the function that will be triggered by the event. The following example shows you how to wire a click event to an element with *myEvent* id in the Customer view (it doesn't exists in the template you previously saw) and wire it to a *doSomething* function:

```
var CustomerView = Backbone.View.extend({
  el: "#customer",
  events: {
    "click #myEvent": "doSomething"
  },
```

```
  template: _.template($("#customer-template").html()),
  render: function () {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
  },
  doSomething: function() {
    console.log("did something");
  }
});
```

There are other aspects of views that this article doesn't cover. For further information, I recommend going to Backbone.js website: http://backbonejs.org.

THE ROUTER

Routing is the most important mechanism in a SPA. In a SPA, you use client-side routing to change views in the SPA shell. Backbone.js includes a *Router* object that exposes routing using the # sign in the URL and HTML5 History API to keep track of the routing. When navigation occurs with the # sign in the URL, the Backbone.js *Router* will intercept the navigation and replace it with calls to route functions, that the developer provided.

When you want to use the Router, you will extend it like all the other Backbone.js objects. The following code shows a simple router:

```
var router = Backbone.Router.extend({
   routes: {
      "": "index",
      "contact": "contact"
   },
   index: function() {
      console.log("index route was triggered");
   },
   contact: function() {
      console.log("contact route was triggered");
   }
});
```

In the example, you can see the use of the *routes* literal object to configure all the routes and all the routing functions. When no route is supplied (for example http://myexample.com/#), the *index* function will be triggered. When someone uses the URL with # followed by contact (for example http://myexample.com/#contact), the *contact* function will be triggered. The *Router* object includes many other aspects such as API and custom routes, that won't be covered in the article, but you can always refer the documentation to learn more.

THE EVENTS API

The Backbone.js Events API is an inner Backbone.js module that all the other Backbone.js objects use for event handling. The API offers functions such as *on, off* and *trigger* that resemble jQuery event model and can help you to wire event handlers to objects. Backbone.js also adds functions like *listenTo* and *stopListening* to enable an object to listen to events, on another object. We won't cover how to use this API but in the example that we covered in The View section; when you used the events literal object, Backbone.js attached the event handler using the Events API.

MAKING BACKBONE.JS OBJECTS WORK TOGETHER

Up till now, you only saw how different Backbone.js objects are defined. Let's take a look at a small example of how to combine the objects in order to render some views. In the example, we are going to use two different templates; one for a customer object and one for customer list:

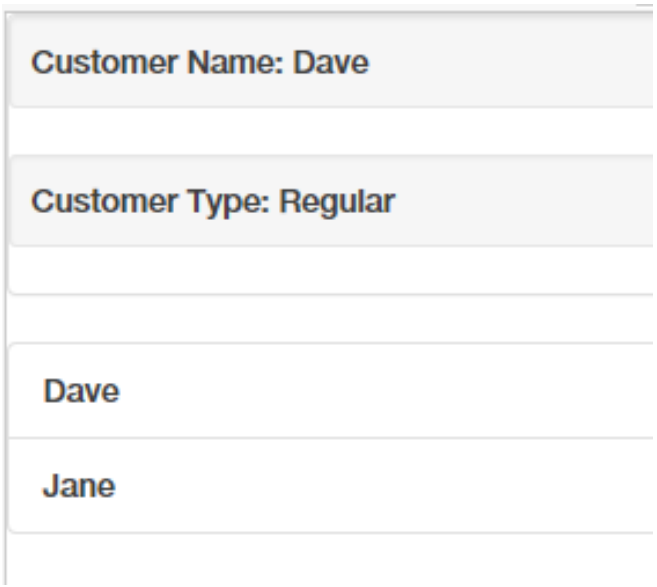
```
<script id="customer-template" type="text/template">
    <div>Customer Name: <%= name %></div>
    <div>Customer Type: <%= type %></div>
</script>
<script id="customers-template" type="text/template">
    <li><%= name %></li>
</script>
<div id="customer"></div>
<ul id="customers"></ul>
```

Once you put this HTML in your web page, add the following code which defines all the Backbone.js objects and runs the *render* functions of the views:

```
var Customer = Backbone.Model.extend({
  defaults: {
    name: "",
    type: "Regular"
  }
});
```

```
var Customers = Backbone.Collection.extend({
  model: Customer
});

var CustomerView = Backbone.View.extend({
  el: "#customer",
  template: _.template($("#customer-template").html()),
  render: function () {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
  }
});

var CustomersView = Backbone.View.extend({
  el: "#customers",
  template: _.template($("#customers-template").html()),
  events: {
    "click li": "changeModel"
  },
  render: function () {
    this.model.each(this.addOne, this);
    return this;
  },
  addOne: function(car) {
    this.$el.append(this.template(car.toJSON()));
  },
  changeModel: function(e) {
    var customersArr = this.model.where({ "name":
    e.currentTarget.innerText});
    var customerView = new CustomerView({ model:
    customersArr[0] });
    customerView.render();
  }
});

// create two customers
var dave = new Customer({
    name: "Dave"
});
var jane = new Customer({
    name: "Jane",
    type: "Platinum"
});
// create the customer collection
var customers = new Customers([dave, jane]);

// create the customers view and bind it to the customer
collection
var customersView = new CustomersView({model:
customers});

// render the customers view
customersView.render();
```

The code is very easy to understand. You define the objects you are going to use by extending Backbone.js objects. Later on, you just bind the Collection to the customers view and render that View. You can also see the Event Handler to change the Model while clicking on each list item in the list. When a list item is clicked, a customer view is created and shows the details of the customer. You can use online JavaScript playground websites such as jsFiddle to run this code and see it in action. The output might look like the following figure:



You can find the code that was used in this section in jsFiddle website: jsfiddle.net/dotnetcurry/hbs4tLnf/

BACKBONE.JS IN REAL-WORLD EXAMPLES

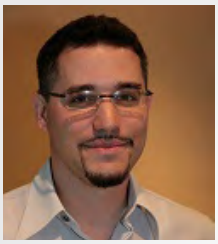
If you are still in two minds about using Backbone.js, just take a look at the list of websites and web applications that use it at: http://backbonejs.org/#examples. You will find that the list comprises of well known websites like USA Today, Airbnb and Khan Academy. You will also find web applications such as Foursquare, Bitbucket and Code School that make use of Backbone.

SUMMARY

Backbone.js is a popular library that helps to impose structure in JavaScript. It also includes features that enables web developers to create SPAs more easily. You can use the exposed Backbone.js objects and their APIs to build common web functionality such as server interaction or DOM manipulation. Above all, Backbone.js has a huge community (*~18500 stars and ~4100 forks in Github as of this writing*).You can also find many Backbone.js plugins created by the community such as Backbone.Marionette and Knockback which can help you boost your development.

On the other hand, Backbone.js is only a library and you will need to use other libraries in order to create a lot of features that aren't exposed by the library. If you like a much more complete SPA framework that imposes conventions, you might want to take a look at AngularJS or Ember. All in all, Backbone.js is very helpful and is used in a lot of known web applications and websites ■

AUTHOR PROFILE



Gil Fink is a web development expert, ASP.Net/IIS Micrsoft MVP and the founder of sparXys. He conducts lectures and workshops for individuals and enterprises who want to specialize in infrastructure and web development. He is also co-author of several Microsoft Official Courses (MOCs) and training kits, co-author of "Pro Single Page Application Development" book (Apress) and the founder of Front-End.IL Meetup. You can get more information about Gil in his website http://www.gilfink.net

IMPORTANCE OF UNIT TESTING

Gardeners understand the problems that insects can cause to their plants. Entire gardens can be destroyed in short time. To stop insects from killing the plants, gardeners use *insecticides*. In fact, different types of insecticides may be needed. Some are organic. Some are chemical. Some kill off one type of insect, others kill off a different type of insect. And, of course, another name for the insect is *bug*.

Bugs kill our Software Garden as quickly as insects kill gardens. Our insecticide is *good testing*. Just as gardens use different types of insecticides, as software gardeners, we should use different types of testing. Integration, capacity, acceptance, functional, system, unit, regression, stress, and performance are just some of the testing types we can use.

Of all these types of testing, there is one in particular that is of interest to developers and is the first line of defense when considering insecticides for your software. That one test type is *unit testing*.

Think for a minute about your code. Do you have statements that include if, switch, for, foreach, while, do…while? Now think about traditional testing techniques where the developer writes the code, compiles it, checks it in, and then throws the executable over the wall to the QA team. They go to work, testing the running program, finding bugs, then throwing things back to the developer, who writes code, compiles it, checks it in, and throws it over the wall again. This cycle continues on and on and on.

Why does this cycle continue? One reason is the QA team has no idea where every if, for, foreach, while, etc. exist in the code. No, scratch that. They don't know where *any* of those statements are in the code because they don't see the code. How can QA possibly know how or where to test the application? But it's worse than you think. In his seminal book, "Testing Computer Software", Cem Kaner talks about G.J. Myers who, in 1976 described a 100-line program that had 1018 unique paths. Three years later, Meyers described a much simpler program. It was just a loop and a few IF statements. In most languages you could write it in 20 lines of code. This program has 100 trillion paths.

Those numbers are daunting. How can you possibly test this? The short answer is, you can't. The long answer is, Meyers had contrived examples, but they do show how complicated code can be and how important it is for the developer to write unit tests.

In this issue, I'm going to show you how to get started with unit testing using a simple ASP.NET MVC application. You'll see how to setup the test and remove the database from the testing process. Through it all, I'll keep it simple. I will not discuss JavaScript testing nor will you see production ready code. I'm also not going to talk about Test Driven Development (TDD) because if you're learning unit testing, you have enough to figure out without turning the entire development process upside down.

INSECTICIDE

WHAT EXACTLY IS UNIT TESTING?

Unit tests are written by the person who wrote the code. Who else knows the code better? Unit tests should be run entirely in memory and should not access databases, nor external files or services. A unit is a small piece of code; for example, a method. The method should be kept small and should do one thing and one thing only. Not only does this make the code easy to test, it makes it easy to maintain.

To write unit tests, you need a unit testing framework. Some popular ones are MSTest that comes with Visual Studio, NUnit, and XUnit. There are others. I will use NUnit in my examples. Don't worry about downloading NUnit. We'll use NuGet to add this to our test project.

You then need a way to run tests. Test runners execute the tests and give feedback about which tests pass and which fail. This is typically called red/green. Red tests fail. Green tests pass. You need two types of runners. The first is some type of GUI application that you use while you are creating code. The other is a console application that you can use on your Continuous Integration (CI) server. Choose the runner that supports the testing framework you have chosen. Again, MSTest is built into Visual Studio, but NUnit and XUnit have their own runners. Additionally, the Visual Studio test runner has been opened up so that it will also run NUnit and XUnit tests. There are also commercial tools, such as Resharper, that have test runners. Then there are commercial tools, such as NCrunch and Test Driven.Net, that are specially designed just for running unit tests.

I'm going to use NUnit using the Visual Studio test runner for the examples, then at the end, I'll show you NCrunch and tell you why it's a superior tool and how increased productivity will pay the cost of the tool.

So, which unit testing framework should you choose? If you are in a Team Foundation Server (TFS) shop, MSTest is probably your best choice. The test runner is built into Visual Studio and TFS. If you are not in a TFS shop, choose one of the others because you can't separate the MSTest runner from Visual Studio or TFS to install on your build server. I use TeamCity from JetBrains as my Continuous Integration server. It has its own test runners for NUnit and MSTest. It also has a built-in test coverage tool for NUnit and supports console application test runners for other unit test frameworks.

VISUAL STUDIO PROJECT SETUP

Before creating our project, let's enable NUnit. We'll do this in two steps. First, install the NUnit Test Adapter. This will allow NUnit to integrate into Visual Studio and use the Visual Studio test runner. To install the NUnit Test Adapter, select Tools > Extensions and Updates from the Visual Studio menu. On the left-hand menu, select Online, then type NUnit into the search box. Click Install and walk through the install process. You'll need to restart Visual Studio when you're done.

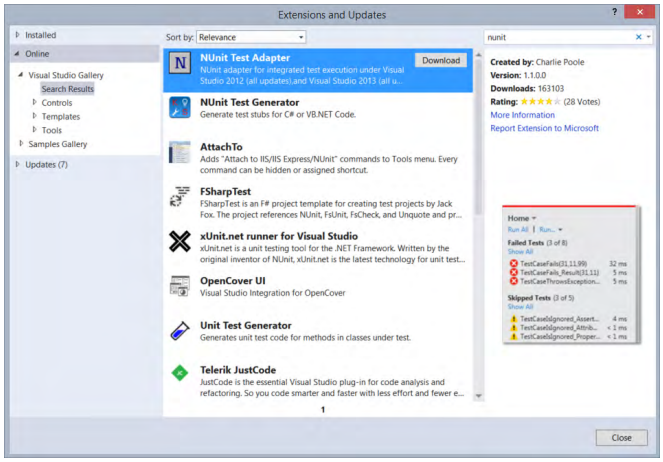


Figure 1: Extensions and Updates dialog in Visual Studio

We'll handle the second part of adding NUnit after creating the project. My example application is an ASP.NET MVC 5 application. When I created it, I named the solution DncDemo and the project DncDemo.Web. I checked Add unit tests and renamed the unit test project to DncDemo.UnitTests. I did not change authentication options. I unchecked Host in the cloud. Once Visual Studio has created the project, we can move on to the second part of NUnit setup; adding the NUnit assemblies. You can do this through the NuGet Package Manager. Be careful that you add it only to the DncDemo.UnitTests project. Add a reference to the DncDemo.Web project.

Go ahead and add a simple Model, Controller, and some Views. I have one named Customer that is for simple CRUD of the Customer table.

In a production application, I have all data access go through a Data project (in this example, it would be named DncDemo.Data) and rename the Models folder to ViewModels. I am not doing that here because I want to keep things simple.

As a best practice, MVC controllers should be very thin, meaning they should not do anything other than pass the request on to another object and then return the result back to the View. In other words, there shouldn't be any data access or business logic in the Controller. In the DncDemo.Web project, add a new folder named Services. The controller will use classes in this folder to handle everything.

YOUR FIRST UNIT TEST

Now, let's create the first unit test. We'll start with something simple, the Index method of the Customer class. The first thing to do is think of something to test. Let's see, we can make sure we get all the rows in the customer table. I'll walk you through the steps.

1. The Index method of the CustomersController has some data access code. Start by refactoring it out. Create a new class named CustomerService in the Services folder. Move code from the CustomersController to the service.

```
public class CustomerService
{
  private Context db = new Context();

  public List<Customer> GetCustomersForIndex()
  {
    return db.Customers.ToList();
  }
}
```

2. Now update the controller to use the service. When you refactor, work slowly, working on small parts of code at a time. Eventually, every method in the controller will be refactored to call the service and you can remove the instantiation of the Context object. I'll leave most of the work for you to do as an exercise.

```
private Context db = new Context();
private CustomerService service = new
CustomerService();

public ActionResult Index()
{
  return View(service.GetCustomersForIndex());
}
```

3. The refactoring is done for now, so you should test the code. Since we don't have any unit tests yet, you'll need to run the site and navigate to the Customer Index page.

4. Now add a new class, CustomerTests to the DncDemo.UnitTests project. Add a reference to NUnit.Framework.

```
using DncDemo.Web.Services;
using NUnit.Framework;

namespace DncDemo.UnitTests
{
    [TestFixture]
    public class CustomerTests
    {
        [Test]
        public void Index_Returns_AllRows()
        {
            // Arrange
            CustomerService service = new
                        CustomerService();

            // Act
            var actual = service.
                        GetCustomersForIndex();

            // Assert
            Assert.AreEqual(3, actual.Count);
        }
    }
}
```

The *TestFixture* attribute tells NUnit that the class contains tests. The *Test* attribute tells NUnit that the method is a test.

The name of the method is important. It should be made up of three parts, what you are testing (Index), what test actually does (Returns), and what the expected result is (AllRows). A test should test for one thing and one thing only. You may have many tests that do nothing but return the results of the GetCustomersForIndex method, each testing for a different result.

Now down in the code, three things are happening. Every good unit test does these three things - Arrange, Act, and Assert. Arrange is where you do all the setup needed to prepare to run the method under test. Act is where you actually call the

method. Assert is where you compare the results of the method with what you expect.

Now that the code has been refactored and the unit test added, it's time to run the test. Here are the steps:

1. Stop writing code

2. On the Visual Studio menu, select Test > Run > All tests

3. Wait for the code to compile

4. Wait for the tests to run

5. View the results

You can see the test results in several places. First in the code for the service.

```
2 references | 0/1 passing
public List<Customer> GetCustomersForIndex()
{
    return db.Customers.ToList();
}
```

Figure 2: Unit test results in the code under test.

Next, in the unit test

```
[Test]
0 references
public void Index_Returns_AllRows()
{
    // Arrange
    CustomerService service = new CustomerService();

    // Act
    var actual = service.GetCustomersForIndex();

    // Assert
    Assert.AreEqual(3, actual.Count);
}
```

Figure 3: Unit test results in the test code.

Finally, in the Test Explorer window.

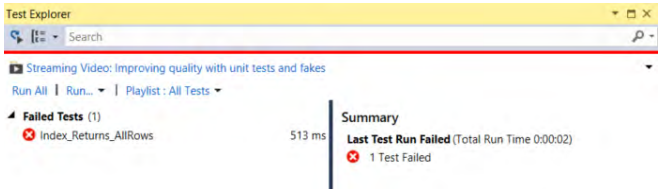


Figure 4: Test Explorer showing failing tests.

In all three places, you see a red circle with an X, indicating the test did not pass. The Test Explorer gives additional information, the time it took to run the test.

So, we know there is an error because the test failed. Can you see it? Can you see other problems? The Assert is explicitly checking that three rows were returned. How can we know there are three rows in the table? The name of the method is Index_Returns_AllRows. Yet, we are checking for three. Finally, the test won't even run. It throws an error because it can't even get to the database.

With all this information, we need to fix something. First, you have to figure out what's not working. In this case, the code can't reach the database because the app.config file for the unit test project doesn't know about the database. Don't add a connection string. Remember, unit tests should run entirely in memory. We need a way to :

1) remove the database access and

2) know how many rows are in the "table" that we query.

REMOVING THE DATABASE

Removing the database is easier than is sounds. We'll do it in two steps, *repositories* and *mocks*. The cool thing is, once we're done, the exact same code will work for unit testing or for when we actually run the web site. This is important. If we have to change code between unit testing and actually running, we could have untested code that contains bugs.

REPOSITORIES

When you look at the code, the CustomerService class instantiates the context. You might be inclined to use IContext instead of DbContext to get rid of the database. But Entity Framework didn't have IContext until version 6 and even then, it's not really suitable for unit testing. We'll fall back to a well-known pattern called the *Repository Pattern*.

In the Models folder, create a new interface called ICustomerRepository. In this interface, you'll define all the methods needed to access the Customer table. You should name each method something that makes sense for the context it's used for. In the code, I've defined four methods even though we'll only implement one of them in this column.

```
public interface ICustomerRepository
{
  IQueryable<Customer> GetAll();
  Customer GetById(int id);
  void Save(Customer customer);
  void Delete(Customer customer);
}
```

Now to implement the interface. Add the class CustomerRepository to the Models folder.

```
public class CustomerRepository : ICustomerRepository
{
  private Context _context = new Context();

  IQueryable<Customer> ICustomerRepository.GetAll()
  {
    return _context.Customers;
  }

  Customer ICustomerRepository.GetById(int id)
  {
    throw new NotImplementedException();
  }

  void ICustomerRepository.Save(Customer customer)
  {
    throw new NotImplementedException();
  }

  void ICustomerRepository.Delete(Customer customer)
  {
    throw new NotImplementedException();
  }
}
```

The Context class is instantiated as a field and then used to get the actual data from the database.

Finally, we need to refactor the CustomerService class to use the repository.

```
public class CustomerService
{
  private ICustomerRepository customerRepo = new
  CustomerRepository();

  public List<Customer> GetCustomersForIndex()
  {
    return customerRepo.GetAll().ToList();
  }
}
```

The unit tests won't run yet, but you can run the web site to verify data is getting returned to the Index method. Don't confuse running the site with running a unit test. Think about what takes less time, running the site, logging in, navigating to the page and then verifying the data or running the unit test.

MOCKING THE DATABASE

Next up, we need to make the test pass. To do this, we will trick the CustomerService class into thinking there really is a database. It's actually quite easy because of the ICustomerRepository interface. To make it easier, we'll use a mock, which is nothing more than a fancy way of faking out the CustomerService class. A mocking framework makes this easier to do. I'll use one called Moq. Using NuGet, add Moq to the DncDemo.UnitTests project. Do not add it to the DncDemo.Web project.

Here's the modified unit test code wired up for the mock.

```
using System.Linq;
using DncDemo.Web.Models;
using DncDemo.Web.Services;
using Moq;
using NUnit.Framework;

namespace DncDemo.UnitTests
{
  [TestFixture]
  public class CustomerTests
  {
    [Test]
    public void Index_Returns_ThreeRows()
    {
      // Arrange
      Mock<ICustomerRepository> mockRepo = new
      Mock<ICustomerRepository>();
      mockRepo.Setup(m => m.GetAll()).Returns(new
      Customer[]
      {
        new Customer {Id = 1, FirstName = "Herman",
```

```
        LastName = "Munster"},
        new Customer {Id = 2, FirstName = "Rocky",
        LastName = "Squirrel"},
        new Customer {Id = 3, FirstName = "George",
        LastName = "Washington"}
      }.AsQueryable());

      CustomerService service = new
      CustomerService(mockRepo.Object);

      // Act
      var actual = service.GetCustomersForIndex();

      // Assert
      Assert.AreEqual(3, actual.Count);
    }
  }
}
```

First, I renamed the test method to Index_Returns_ThreeRows() to indicate the number of rows returned. This better describes the expected results. Next, in the Arrange section, the mock is instantiated. And then the mock is setup. What that line says is when the GetAll method is called, the return value is an array of three items that is cast to Queryable. In the Act section, note that I changed the instantiation of the CustomerService class to pass an ICustomerRepository to the constructor. So, now we need to fix up that class.

```
public class CustomerService
{
  private readonly ICustomerRepository customerRepo;

  public CustomerService()
  {
    this.customerRepo = new CustomerRepository();
  }
  public CustomerService(ICustomerRepository
  customerRepo)
  {
    this.customerRepo = customerRepo;
  }
  public List<Customer> GetCustomersForIndex()
  {
    return customerRepo.GetAll().ToList();
  }
}
```

What will happen for testing is the mocked ICustomerRepository will be passed in and used. At runtime, CustomerRepository is instantiated using the constructor with no parameters, so it will use the actual database.

Now repeat the same five steps for running the unit test. If you look at Test Explorer, you'll see the test now passes.

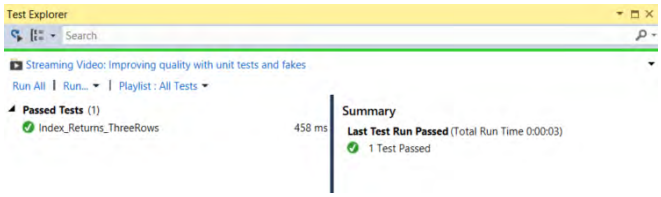


Figure 5: Test Explorer showing passing tests.

This is a common pattern when working with unit tests: Write code, write tests, run tests, see the fail, refactor, update tests, run tests. It's a great feeling when tests pass. It gives better confidence the code is correct and you'll have fewer bugs.

A BETTER WAY TO UNIT TEST: NCRUNCH

Go back and look at the steps for running unit tests. To summarize, you stop writing code, you wait for the project to compile, you wait for unit tests to run. I can't stress enough the negative impact this has on productivity. Basically, you're sitting at your desk, waiting for tests to complete. But there is a better way, NCrunch!

At first, you may balk at spending money on a commercial application, but I assure, it will quickly pay for itself due to increased productivity. This is because NCrunch compiles code in the background and displays results right in Visual Studio, line by line. No stopping the code writing process. No waiting for the code to compile. No waiting for tests to run. This is one tool every software gardener needs to have in the toolshed.

Download NCrunch from http://www.ncrunch.net then install it. Open your project and enable NCrunch for the project. From the Visual Studio menu select NCrunch > Enable NCrunch. You may have to walk through the NCrunch Configuration Wizard (an option on the NCrunch menu). When I do this, I usually pick the defaults on every page except Ignored Tests, where I generally select *Let my tests run*.

Once you finish the wizard, you'll notice some black dots along the left edge (called the gutter) of the Visual Studio editor.

These appear in both the actual code and the unit test code. The first time, you have to enable the test. Go to the unit test code, right click on the green circle with the line through it and select Unignore starting test. You will immediately see some of the black dots turn to green. This tells you the tests passed. Switch to the CustomerService code. The dots are there too!

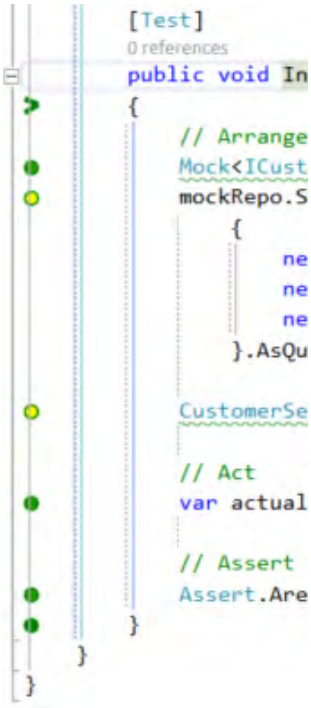


Figure 6: Unit test results with NCrunch.

Yellow dots indicate the line of code that took longer to run than NCrunch thought it should. You may need to see if you can optimize that code. If the dots are red, the test doesn't pass. A red X indicates the actual line of code that failed.

Green, red, and yellow dots also tell you something else. You immediately get a feel for code coverage. This is an important unit testing concept that tells you which lines of code have tests against them and which ones don't. The black dots indicated untested code. To get code coverage with the Visual Studio test runner you have to stop coding, select the Code Coverage option from the VS Test menu, then wait for code to compile and tests to run.

If you now make a change to CustomerService.GetCustomersForIndex or the unit test code, NCrunch will do its work in the background and give you immediate feedback.

NCrunch also has features such as debugging into failing tests, code metrics, support for both NUnit and MSTest frameworks, and many more. I won't go into these features now. That's an exercise for you.

Note that NCrunch is a Visual Studio add-in so you still need a way to run unit tests on the build server. That's where something like TeamCity's unit test runner, TFS, or your unit test framework's console runner come into play.

WHERE TO NEXT?

One test does not complete your testing job. Try to think of other tests to run. For example, what happens if there are no rows returned? Or null? Or you can test to verify the object really is IQueryable<Customer>.

Unit tests should look at two main areas. The first, called the happy path, the code works correctly. The second, the sad path, tests what happens when things don't go right. Does the application blow up? Is a specific error thrown (unit test frameworks let you check for this). How about bounds checking? A common thing to check for is null. It's surprising how many applications don't properly handle null.

One question I often get asked is, "How do I integrate this into legacy code?" There are three candidates for code that should get unit tests. First, code you are modifying. Second, code that has lots of bugs or customers complain about most. Third, areas of code that you are afraid to modify because it's fragile and breaks every time you change it.

Don't be afraid to tell your boss that doing unit testing will add time to write the code. But that time will be far less than writing the code, then fixing bugs that come back from the QA team. And the QA team will spend less time testing the code and more time doing QA activities that you can't easily automate.

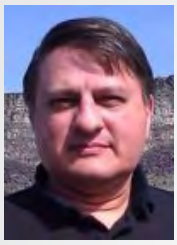
Work slowly, refactor to small, easily tested methods. Write the tests as you go. Don't move onto another section until the section you're working on passes all its tests. If you have a particularly bad section of code, you might want to set aside half or even a full day a week for refactoring and writing unit tests until you're satisfied it's fixed. And don't check the code into Version Control until all unit tests pass.

There is much more to unit testing and mocks. I encourage you to explore those areas. You may also consider learning about Dependency Injection. It will eliminate the need for two constructors in the service class. Finally, once you get proficient with unit testing, you may want to look at Test Driven Development (TDD). What I've shown you here is Test After Development (TAD) where you write the code then write the tests for it. With TDD, you write the test first, then write code to make it pass. It not only tests the code, but tests your assumptions about how the code should be architected.

SUMMARY

Unit tests are one of the most important tools you can use. They are your first line insecticide against bugs that can kill your application. Every application you write, should use them. By following the gardening practices here, your software will grow and be lush, green, and vibrant ■

AUTHOR PROFILE



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.

USING REST IN SHAREPOINT APP TO PERFORM CRUD OPERATIONS

SharePoint is a web application framework that enables you to develop Content-driven, Scalable and Collaborative applications. In SharePoint 2013, Microsoft has fundamentally changed how developers can create web applications using the new cloud app model.

SOME IMPORTANT FEATURES ABOUT THE SHAREPOINT 2013 APP

- SharePoint App is a single package of functionality which can be deployed and activated on-site and has less overhead on a SharePoint Server Farm.

- This package contains all the SharePoint artifacts like Lists, Libraries, etc. along with Pages, Scripts, CSS etc. that are used for an application.

- Apps can be easily deployed on the site and can also be easily removed when they are no longer required.

- Apps does not use any Server-Side Object Model (SSOM) in the code, hence the code does not create any unmanageable code on the server farm and does not corrupt memory.

Additional information about Apps for SharePoint can be found here:
http://msdn.microsoft.com/en-us/library/office/fp179930(v=office.15).aspx

A SharePoint App can be designed using Visual Studio 2012 Ultimate with update 4 or Visual Studio 2013 Ultimate (recommended use Update 2). SharePoint Apps can be deployed on SharePoint online (http://portal.microsoftonline.com) or using on-premise infrastructure. If on-premise infrastructure is used, then the required configuration laid down by Microsoft is a must for the development and deployment of the app. The steps for configuring an environment for Apps can be found from the following link.
http://technet.microsoft.com/en-us/library/fp161236(v=office.15).aspx

HISTORY OF THE SHAREPOINT API PROGRAMMING MODEL

SharePoint technology is already very popular and several professionals and enterprises have built many applications using the Object Model (OM) provided by it. Developers have been able to build Web parts and have access to the SharePoint's Server-Side Object Model (SSOM) that allows them to call SharePoint data and perform operations programmatically. This feature has helped them to build all kinds of solutions.

In SharePoint 2010, if you had to customize or add new features to SharePoint, the only way was to reluctantly install code (which could be untrusted) directly into SharePoint's servers. Although sandbox solutions existed, the restrictions applied were stringent which forced developers to run even untrusted custom code, in full-trust mode. Some of the issues of running fully trusted code in SharePoint 2010 was it could destabilize the whole farm and it was a challenge to migrate to newer versions of SharePoint. Since the code made use of SSOM and ran as a part of SharePoint Processes, there was possibility of high memory consumption and CPU utilization. So to manage the code well, it became an overhead for developers to know about SSOM objects and dispose them appropriately to reduce memory utilization.

HOW DOES 'REST' FIT IN SHAREPOINT PROGRAMMING?

In the earlier versions of SharePoint, SOAP based web services were provided for performing data operations, but ultimately it was only SSOM that was used to write programs with SOAP. This lead to bulky data transfers as a result of SOAP formatted XML payload. Because of this drawback (and a couple of others) of SSOM and SOAP based web services, the developers' community was provided with a new set of APIs for developing SharePoint solutions and integrating them with various types of clients.

In SharePoint 2010, developers were provided with Client Side Object Model (CSOM) using which the ability of accessing SharePoint functionality was made possible in remote client applications like Silverlight.

CSOM was made available in the following flavors:

- Managed Code (.NET)
- Silverlight
- ECMA Script (actually JavaScript)

The underlying protocols that the CSOM used to talk to SharePoint were XML and JSON (JavaScript Object Notation) so that other systems that could make HTTP requests could make calls to SharePoint Data. This was a major feature because code created with CSOM was running remotely. *One limitation of CSOM was that it has provided API support for SharePoint Foundation like Sites, Lists and Documents.*

In SharePoint 2013, a new API support has been added along with the one provided in SharePoint 2010. This provides access to its APIs in several forms, to help development of remote applications using REST/OData for .NET as well as for working with frameworks other than .NET. With this new set of APIs, if developers want to use client applications not using managed code or Silverlight, they can consider the use of REST/OData endpoints to read/write data from and to SharePoint. Since the REST interface doesn't require any reference to assemblies, it also allows you to manage and limit the footprint of your Web applications; an important consideration especially when you are building mobile apps for Windows Phone written using HTML and JavaScript. One of the biggest advantage is that we can make use of JavaScript libraries like, jQuery, Knockout, Angular, etc to build applications, ultimately making use of the skillsets most developers are equipped with.

The APIs endpoints can be accessed using _API. This is a single endpoint for all remote APIs provided in SharePoint 2013. _API is fully REST and OData enabled. Some example url's are as shown here:

http://HostServer/sites/MyDeveloper/_api/web/Lists
http://HostServer/sites/MyDeveloper/_api/web/Lists/getbytitle('MyList')/

Using REST in SharePoint App

In SharePoint 2013 with SharePoint Hosted Apps, we often need to access SharePoint List data for performing CRUD operations. Since SharePoint Apps are programmed using JavaScript and we have a REST API with us, it is possible to use Ajax methods to make calls to the SharePoint List.

The scenario discussed here is that the SharePoint List is available on the site collection. This list can be accessed using SharePoint App for performing CRUD operations. But there exists an isolation between List and the SharePoint App. So in this type of a scenario, our SharePoint App must have access permissions to the specific list on which operations need to perform.

The SharePoint App JavaScript code can locate the List in the site collection using the following url approaches:

(1)
```
var url = SPAppWebUrl + "/_api/SP.AppContextSite(@target)" + "/web/lists/getbytitle('" + listName + "')/items?" + "@target='" + SPHostUrl + "'"
```

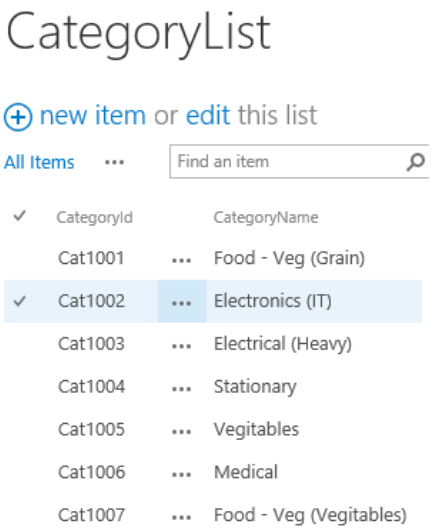
In the above code, the *SPAppWebUrl* represents the SharePoint App URL where the App is hosted and *SPHostUrl* represents the URL for the Site collection from where the List is accessed. The *listName* is the list on which operations are to be performed.

(2)
```
var url = _spPageContextInfo.siteAbsoluteUrl + "/_api/web/lists/getbytitle('" + listName + "')/items"
```

In the above code, the *_spPageContextInfo* is the JavaScript or jQuery variable that provides properties used during SharePoint App development for locating the SharePoint Site collection for performing Ajax calls.

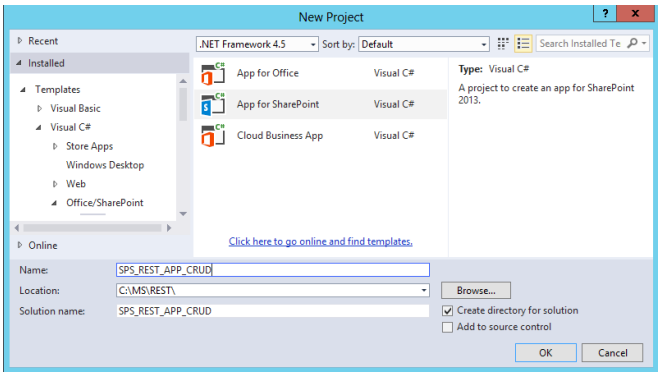
*(**Note:** In this article we have used the second approach)*

For performing operations, we will use the SharePoint 2013 Developer site with the List name 'CategoryList' as shown here:

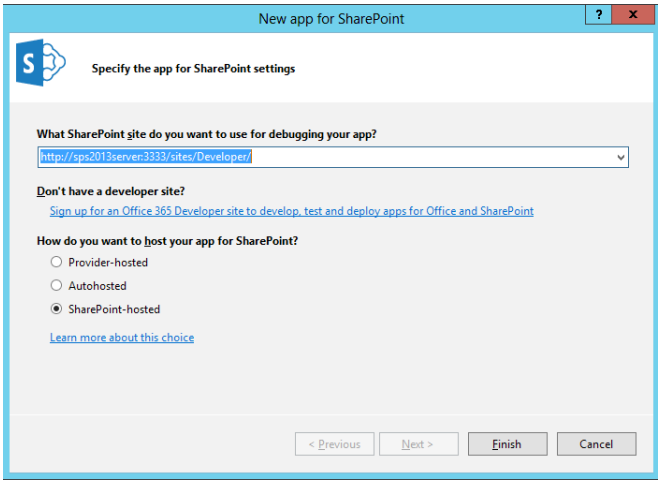


For the implementation, we will use jQuery and Knockout.js. More information about jQuery and Knockout.js, can be obtained from the following links: www.jQuery.com and www.knockoutjs.com

Step 1: Open Visual Studio 2012 Ultimate with Update 4 or Visual Studio 2013 Ultimate (recommended Update 2) and create a new SharePoint App application with the name 'SPS_REST_APP_CRUD':



After clicking 'OK', in the 'Specify the app for SharePoint settings' window, set the site name and hosting option for the app as shown here:



Step 2: From the Scripts folder of the project, open App.js and remove the already available code from it. Now add the following JavaScript which will read the parameters of the URL to get the Host Web Url and App Web Url:

```
'use strict';

var SPHostUrl;
var SPAppWebUrl;
```

```
// read URL parameters
function retrieveQueryStringParameter(param) {
    var params = document.URL.split("?")[1].split("&");
    var strParams = "";
    for (var i = 0; i < params.length; i = i + 1) {
        var singleParam = params[i].split("=");
        if (singleParam[0] == param) {
            return singleParam[1];
        }
    }
}

SPAppWebUrl = decodeURIComponent
(retrieveQueryStringParameter ("SPAppWebUrl"));
SPHostUrl = decodeURIComponent
(retrieveQueryStringParameter ("SPHostUrl"));
```

In the same JavaScript file, create a function which will define observables and functions to act as a *ViewModel* so that it can be used for DataBinding on the UI elements which we will be adding to Default.aspx.

```
//The View Model used for
//1. Defining the ListName
//2. Defining observables used for databinding with UI
//3. getCategories () => Make call to SharePoint List
to Read Categories Data
//4. createCategory() => Define the Data object to be
passed to SharePoint List to Add a new Record in it
//5. getCategoryById() => Retrieve a specific record
from the SharePoint List based upon the Id
//6. updateCategory() => Update a specific Crecord
retrived using getCategoryById() function
//7. deleteCategory() =>  Delete a specific Crecord
retrived using getCategoryById() function

var crudViewModel = function () {
  var self = this;
  var listName = "CategoryList";

  self.Id = ko.observable();
  self.Title = ko.observable();
  self.CategoryName = ko.observable();

  self.Category = {
    Id: self.Id,
    Title: self.Title,
    CategoryName:self.CategoryName
  };

  self.Categories = ko.observableArray();
  self.error = ko.observable();

  //Function to Read all Categories
  self.getCategories = function() {
    $.ajax({
      url: _spPageContextInfo.siteAbsoluteUrl + "/_
      api/web/lists/getbytitle('" + listName + "')/
      items",
      type: "GET",
      headers: { "Accept": "application/
      json;odata=verbose" },
      success: function (data) {
      self.Categories(data.d.results);
    },
      error: function (data) {
      self.error("Error in processing request " +
      data.success);
    }
  });
};

  //Function to Create  Category
  self.createCategory = function () {
    var itemType = "SP.Data.CategoryListListItem";
    var cat = {
      "__metadata": { "type": itemType },
      "Title": self.Title,
      "CategoryName": self.CategoryName
    };

    $.ajax({
      url: _spPageContextInfo.siteAbsoluteUrl + "/_api/
      web/lists/getbytitle('" + listName + "')/items",
      type: "POST",
      contentType: "application/json;odata=verbose",
      data: ko.toJSON(cat),
      headers: {
        "Accept": "application/json;odata=verbose", //
        return data format
        "X-RequestDigest": $("#__REQUESTDIGEST").val()
      },
      success: function (data) {
        self.error("New Category Created
```

```
        Successfully");
      },
      error: function (data) {
        self.error("Error in processing request " +
        data.status);
      }
    });
    self.getCategories();
  };

  //Function to get Specific Category based upon Id
  function getCategoryById  (callback) {
      // Code removed for brevity. Download source
      // code at the end of this article.
  };

  //Function to Update Category
  self.updateCategory = function () {
    getCategoryById(function (data) {
          // Code removed for brevity
    });
  };

  //Function to Delete Category
  self.deleteCategory = function () {
    getCategoryById(function (data) {
         // Code removed for brevity
    });
  };

  //Function to Select Category used for Update and
  Delete
  self.getSelectedCategory = function (cat) {
        // Code removed for brevity
  };

  //Function to clear all Textboxes
  self.clear = function () {
    alert("Clear");
    self.Id(0);
    self.Title("");
    self.CategoryName("");
  };

};
```

In the above code, the *getCategories()* defines 'headers' as `"Accept": "application/json;odata=verbose"`. This represents the format for the return data from the call made to the SharePoint service. The function *createCategory()* defines the data to be saved in the list as shown here:

```
var itemType = "SP.Data.CategoryListListItem";
  var cat = {
    "__metadata": { "type": itemType },
    "Title": self.Title,
    "CategoryName": self.CategoryName
  };
```

The *itemType* variable is set to the value as SP.Data. CategoryListListItem because the name of the List is *CategoryList*. So keep one thing in mind that if you want to use the above logic for various SharePoint Lists, then the value for 'itemType' variable must be generalized. This value is then passed with '__metadata'. This helps to tag the structural content so that before performing write operations on the SharePoint object, the data is checked. The *updateCategory()* and *deleteCategory()* functions makes call to *getCategoryById()* functions to perform respective operation. The headers information to be passed for the Update is as shown here:

```
headers: {
  "Accept": "application/json;odata=verbose",
  "X-RequestDigest": $("#__REQUESTDIGEST").val(),
  "X-HTTP-Method": "MERGE",
  "If-Match": data.d.__metadata.etag
},
```

The header information sets the HTTP method to 'Merge' which checks the record's availability and if the record is found, then it is updated. Similar values are also set in headers during the delete operation; the only difference is the Http Method is specified as 'Delete' as shown here:

```
headers: {
  "Accept": "application/json;odata=verbose",
  "X-RequestDigest": $("#__REQUESTDIGEST").val(),
  "X-HTTP-Method": "DELETE",
  "If-Match": data.d.__metadata.etag
        },
```

Step 3: Open Default.aspx and add the HTML Table and other UI elements with databind expressions for observables and functions declared in ViewModel as shown here: (Yellow

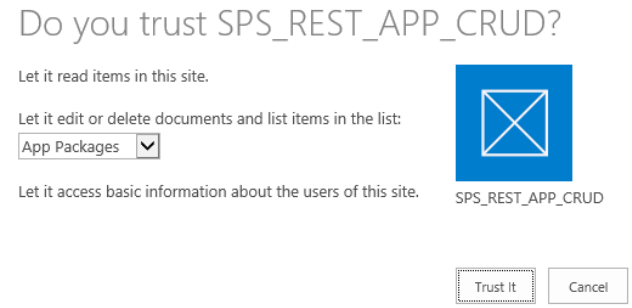
marked.)

```
<asp:Content ContentPlaceHolderID="PlaceHolderMain"
runat="server">

  <table id="tbl">
    <tr>
      <td>
        <table>
          <tr>
            <td>Id</td>
            <td><input type="text" id="txtid" data-
            bind="value: $root.Id"/>
            </td>
          </tr>
          <tr>
            <td>CategoryId</td>
            <td><input type="text" id="txtcatid" data-
            bind="value: $root.Title"/>
            </td>
          </tr>
          <tr>
            <td>CategoryName</td>
            <td><input type="text" id="txtcatname"
            data-bind="value: $root.CategoryName"/>
            </td>
          </tr>
          <tr>
            <td><input type="button" id="btnnew"
            value="New" data-bind="click: $root.clear"
            /></td>
            <td><input type="button" id="btnsave"
            value="Save" data-bind="click: $root.
            createCategory"/>
            </td>
            <td><input type="button" id="btnupdate"
            value="Update"  data-bind="click: $root.
            updateCategory"/>
            </td>
          </tr>
        </table>
      </td>
      <td>
        <div  class="Container">
        <table class="table">
          <thead>
          <tr>
            <th>Id</th><th>Category Id</th><th>Category
            Name</th>
            </tr>
            </thead>
          <tbody data-bind="foreach: Categories">
            <tr data-bind="click: $root.
            getSelectedCategory">
              <td>
                <span data-bind="text:Id"></span>
              </td>
              <td>
                <span data-bind="text: Title"></span>
              </td>
              <td>
                <span data-bind="text: CategoryName">
                </span>
              </td>
              <td>
                <input type="button" value="Delete"
                data-bind="click: $root.deleteCategory"/>
              </td>
            </tr>
          </tbody>
        </table>
      </div>
    </td>
  </tr>
</table>

<div>
  <span data-bind="text:error"></span>
</div>

</asp:Content>
```

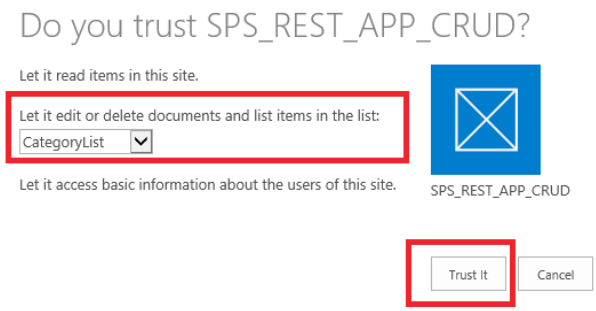
Step 4: Since we are performing Read/Write operations on the SharePoint List, we need to set the concerned permissions for the app. To set these permissions, double-click on the AppManifest.xml and from the Permissions tab, add the permissions as shown here:

Scope	Permission	Properties
List	Write	
Web	Read	
*		

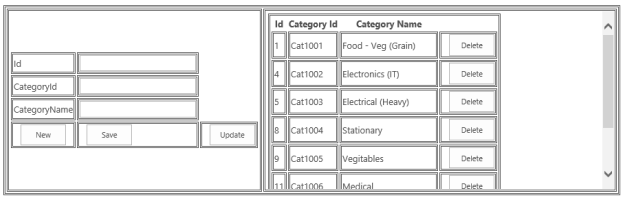
Step 5: Build the App and if it is error free, then Deploy. Once the deployment is successful, the trust page will be displayed as shown here:



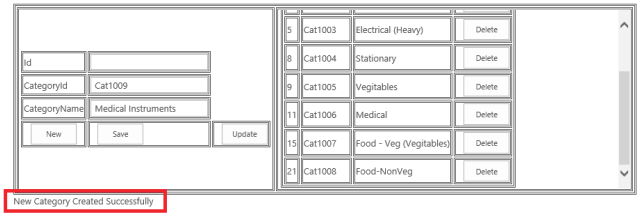
Since we are performing operations on the CategoryList; from the ComboBox, select the CategoryList and click on the 'Trust It' button as seen here:



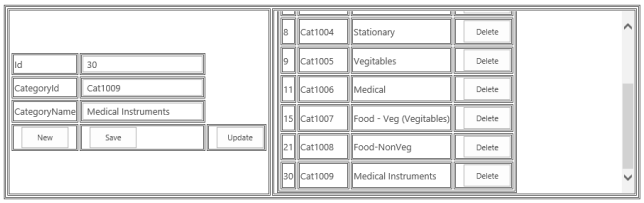
This will bring up the credentials window. Enter Credentials and the App will be displayed with Category List data:



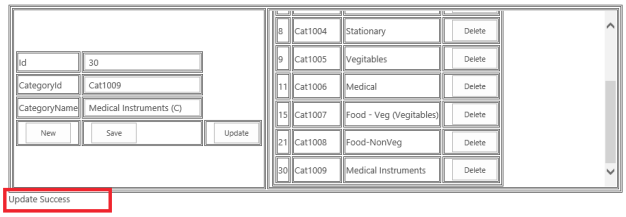
Add values for CategoryId and CategoryName (Id gets generated automatically) and click on the 'Save' button. The Record message gets displayed as seen here:



If any row from the table showing the categories is clicked, then that record gets displayed in the textboxes:



Now this record can be updated and deleted.



Similarly, you can also test the Delete functionality.

CONCLUSION

The SharePoint 2013 REST interface is a powerful approach that provides most functionalities that Web and Mobile app developers seek for. It makes it really simple to work with SharePoint data using client side JavaScript libraries and frameworks. This opens up the SharePoint 2013 development platform to standard Web technologies and languages ■

Download the entire source code from our GitHub Repository at bit.ly/dncm14-sharepointrest

AUTHOR PROFILE

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

THE ABSOLUTELY AWESOME

.NET INTERVIEW BOOK

SUPROTIM AGARWAL

PRAVIN DABADE

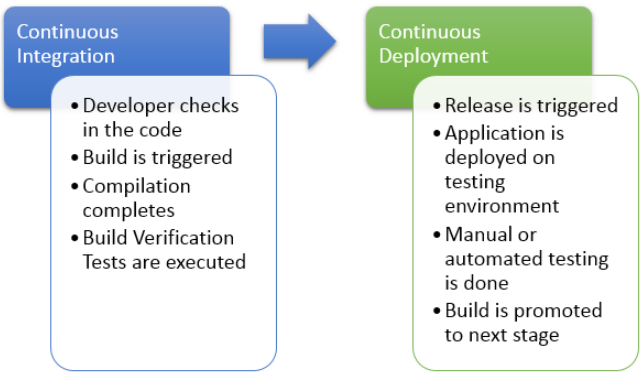
Microsoft Team Foundation Server 2013 provides features for implementing both Continuous Integration (CI) and Continuous Deployment (CD).

Continuous Integration, Continuous Deployment with Team Foundation Server 2013

One of the development practices that agile teams follow is *Continuous Integration*. A check-in into the source control by a developer needs to be ensured to integrate with earlier check-ins, in the same project, by other developers. This check of integration is done by executing a build on the latest code including the latest check-in done by the developer. Agile teams also should follow the principle of keeping the code in deployable condition. This can be checked by doing a deployment of the application to a testing environment after the build and running the tests on it, so that integrity of the recent changes is tested. Those tests can be either manual or automated. If we can automatically deploy the application after the build, then the team has achieved *Continuous Deployment*. It can be extended further to deploy the application to production environment after successful testing.

Microsoft Team Foundation Server 2013 provides features for implementing both Continuous Integration (CI) and Continuous Deployment (CD). For my examples, I have used TFS 2013 with Visual Studio 2013 Update 2.

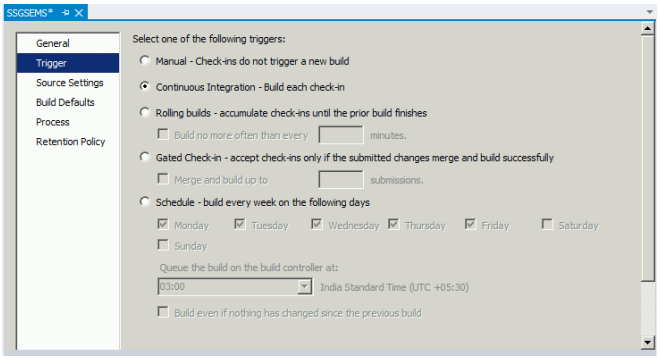
Let us first see the flow of activities that form these two concepts:



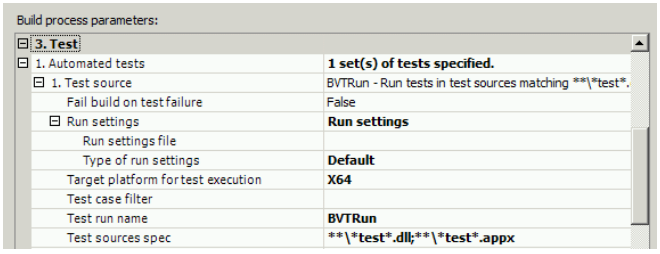
TFS supported CI right from the beginning, when it came into existence in 2005. It has a feature for build management that includes build process management and automated triggering of the build. There are number of automated build triggers supported by TFS. Scheduled build is one of them but for us, more important is Continuous Integration trigger. It triggers a build whenever a developer checks in the code that is under the scope of that build.

Let us view how it is to be configured and for this example, I have created a solution named SSGS EMS that contains many projects.

Entire solution is under the source control of team project named "SSGS EMS" on TFS 2013. I have created a build definition named SSGSEMS to cover the solution. I have set the trigger to Continuous Integration so that any check-in in any of the projects under that solution, will trigger that build.



I have created a test project to store Unit tests of a project. The unit tests project name is 'PasswordValidatorUnitTestProject' which has a unit test to test the code that is under the project "SSGS Password Validator". I wanted to run that unit test as part of the build after the projects are successfully compiled. It is actually a standard setting in Team Build to run any tests under the test project that is under the solution that is in the scope of that build. Just to ensure, I open the Process node of the build definition in the editor and scroll down to the Test node under it.



I added only the value to the parameter *Test run name*. I gave it a name BVTRun, a short form for Build Verification Tests Run.

Now I am ready to execute that build. I edited a small code snippet under the project "SSGS Password Validator" and checked in that code. The build triggered due to that check-in and completed the compilation successfully as well as ran that test to verify the build. I have validated that the Continuous Integration part of TFS is working properly.

Microsoft included deployment workflow recently in the services offered by Team Foundation Server. That feature is called *Release Management*. It offers the following functionality:

To implement Release Management and tie it up with TFS 2013 Update 2, we have to install certain components:

Release Management Server	Release Management Client	Microsoft Deployment Agent
• Centralized repository of RM data • Services to configure RM • Needs SQL Server 2008 onwards as prerequisite • Can be installed on Windows Server 2008 onwards	• Rich UI for accessing RM Server • Can be installed on Windows 7 SP1 onwards or Windows Server 2008 R2 onwards	• Works as agent of RM on the machines where application is to be deployed • Can be installed on Windows Vista onwards

You may find details of hardware and software prerequisites over here http://msdn.microsoft.com/en-us/library/dn593703.aspx. I will guide you now to install these components of Release Management and use them to configure releases.

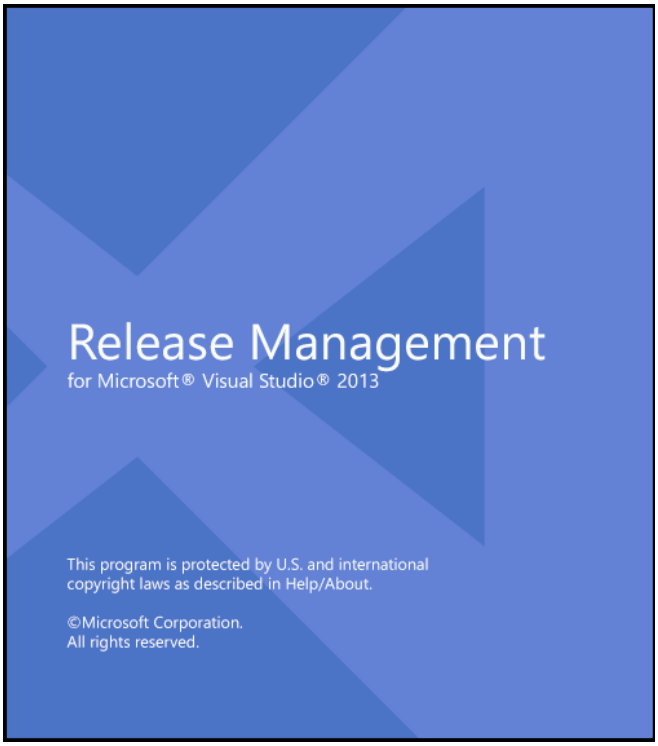
INSTALL AND CONFIGURE RELEASE MANAGEMENT

RELEASE MANAGEMENT COMPONENTS

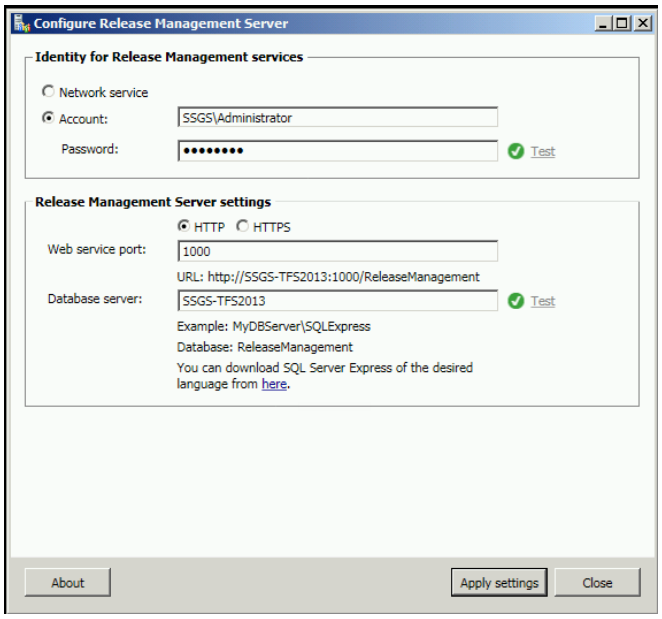
1. **Release Management Server 2013** – Stores all the data in a SQL Server Database. Install this on any of the machines that is in the network with TFS and with those machines where you want to deploy built application. It can be installed on TFS machine itself.

2. **Release Management Client 2013** – Used to configure the RM Server and to trigger the releases manually when needed. Install this on your machine. It can be also installed on TFS.

3. **Microsoft Deployment Agent 2013** – These need to be installed on machines where you want to deploy the components of the built application.



While installing these components, I provided the account credentials which has administrative privileges in the domain so that it can do automatic deployment on any of the machines in the domain. I suggest to keep the Web Service Port as 1000 if it is not being used by any other service.



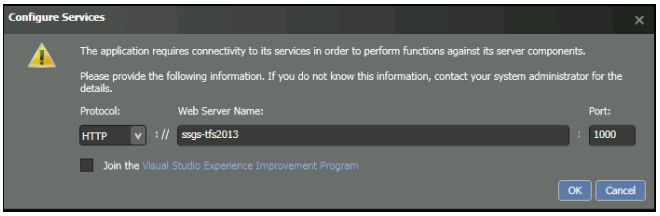
In Addition to these components, if you need to execute automated tests as part of your deployment process, then you also need to install Test Controller and configure Lab Management.

Once the installation is over, let us do the setup of the release management so that releases can be automated on that. Most important part of that setup is to integrate TFS so that build and release can be used hand in hand.

INITIAL SETUP OF RELEASE MANAGEMENT

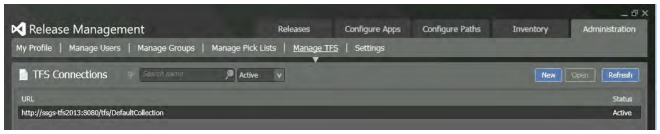
To configure the settings of Release Management, open the RM Client application and select Administration activity.

Let us first connect RM Client with RM Server. To do that, select Settings. Enter the name of RM Server in the text box of Web Server Name. Click OK to save the setting.



Now we will connect RM with our TFS. Click on the Manage TFS link. Click the New button and enter the URL of TFS with

Collection name. Click OK to save the changes.



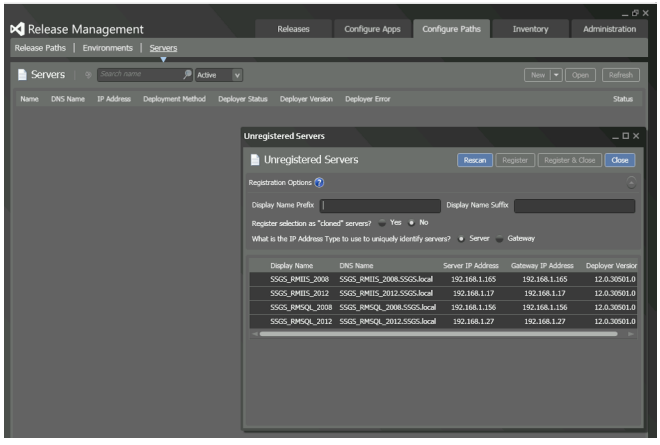
Next step is to add the names of supported stages of deployment and technologies. Generally there are at least two stages – Testing and Production. Sometimes there are more stages like Performance Testing, UAT etc. For specifying the stages of our releases, click the *Manage Pick Lists* link and select *Stages* and enter stages like Development, Testing and Production. Similarly select *Technologies* and enter the names like ASP.NET, .NET Desktop Applications, WCF Services etc.

You can now add users who can have different roles in the Release Management by clicking on *Manage Users*. You may add groups to represent various roles and add users to those roles. These users will be useful when you want to have a manual intervention in the deployment process for validation and authorization.

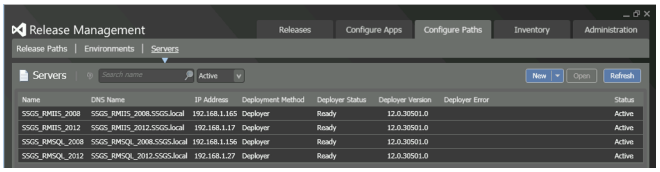
Each release follows a certain workflow. Path of that workflow needs to be defined in advance. Let us now find out how to define that release path.

CONFIGURE RELEASE MANAGEMENT PATHS

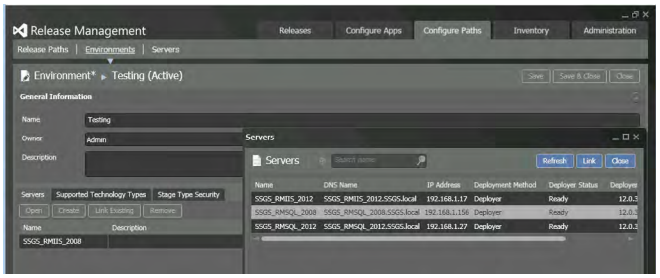
To start building the RM paths, we will need to register the building blocks that are the servers on which the deployment will be done. To do that, first click on the *Configure Paths* tab and then on the *Servers* link. RM Client will automatically scan for all the machines in the network that have the Microsoft Deployment Agent 2013 installed. Since this is first time that we are on this tab, it will show the list as *Unregistered Servers*.



You can now select the machines that are needed for deployment in different stages and click on the *Register* button to register those servers with RM Server.



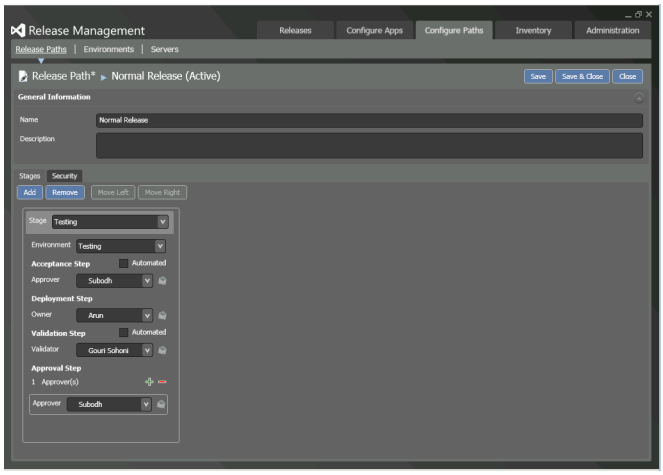
Let us now create the environments for different Stages. Each environment is a combination of various servers that run the application for that stage. Let us first create a Testing environment made up of two of the machines. One of them, an IIS Server that will host our application when deployed, and another a SQL Server for deploying. Provide the name of the Environment as *Testing* (although there may be a stage with similar name, they are not automatically linked). Link the required machines from the list of servers – select the server and click on the link button.



You may create another environment named *Production* and select other servers for that environment.

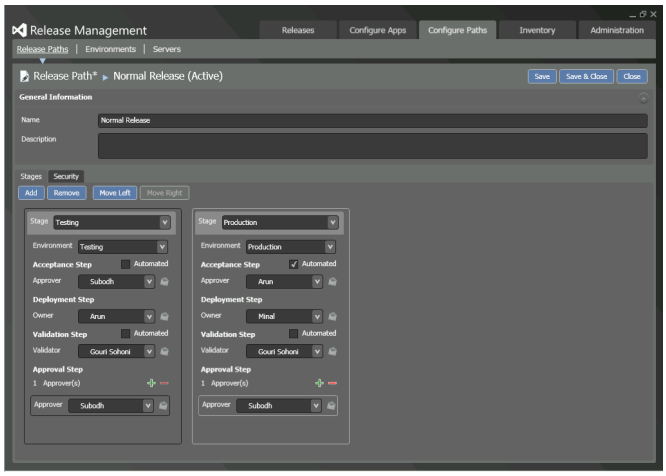
Next step is to configure *Release Paths* that will be used by our releases. These paths are going to use various entities that we have created so far like the stage, environments and users if necessary.

Click on the *Release Paths* under *Configure Paths* activity. Select the option to create new release path. Give it a Name, Normal Release and a Description if you want. Under the stages tab, select the first stage in our deployment process i.e. Testing. Under that stage, now select the environment Testing that will be used to deploy the build for testing.



Set the other properties so that Acceptance and Validation Steps are 'Automated'. Set the names of users for *Approver* for Acceptance step, *Validator and Approver* for Approval step. Approver is the person who decides when the deployment to next stage should take place. It is a step where release management workflow depends upon manual intervention.

Click on the Add button under stages and add a stage of Production. Set properties similar to Testing stage.



Save and close the Release Path with a name. For example I have named it as *Normal Release*.

CREATE RELEASE TEMPLATE

Now we will tie all ends to create a release template that will be used to create multiple releases in the future. In each release in each stage, we want to do deployment of components of our application, run some tests either automatically or manually and finally approve the deployment for next stage. Let us take an example where we will design a release template that has following features:

1. It uses two stages – Testing and Production

2. In each stage, it deploys a web application on a web server and a database that is to be deployed on a database server.

3. It has to run an automated test on the web application after the deployment on Testing environment. It may also allow tester to run some manual tests.

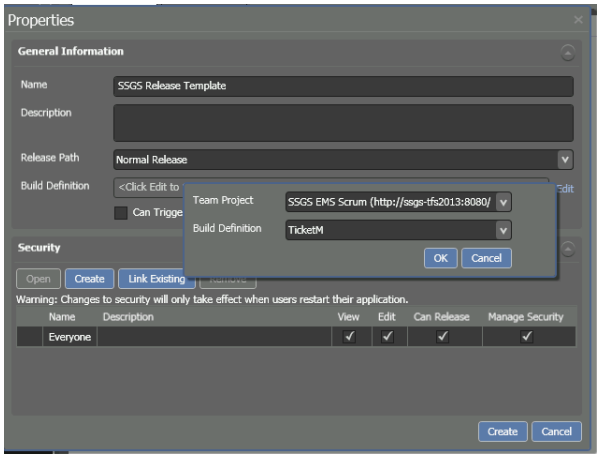
4. It should have an approval from a test lead after testing is successful for the application to be deployed on Production environment.

Set properties of Release Template

Release template is the entity where we will bring together the Release Path, Team Project with the build that needs to be deployed and the activities that need to be executed as part of the deployment on each server. Release template can use many of the basic activities that are built in and can also use many package tools to execute more complex actions.

First select the Configure App activity and then from Release Template tab, select New to create a new release template.

As a first step, you will give template a name, optionally some description and the release path that we have created in earlier exercise. You can now set the build to be used. For that, you will have to first select the team project that is to be used and then the build definition name from the list of build definitions that exist in that team project. You also have the option to trigger this release from the build itself. Do not select this option right now. We will come back to that exercise later.



CONFIGURE ACTIVITIES ON SERVERS FOR EACH STAGE

Since we have selected the configured path, you will see that the stages from that path are automatically appearing as part of the template. Select the Testing stage if it is not selected by default. On the left side, you will see a toolbox. In that tool box there is a *Servers* node. Right click on it and select and add necessary servers for this stage. Select a database server and an IIS server.

Now we can design the workflow of deployment on the stage of Testing.

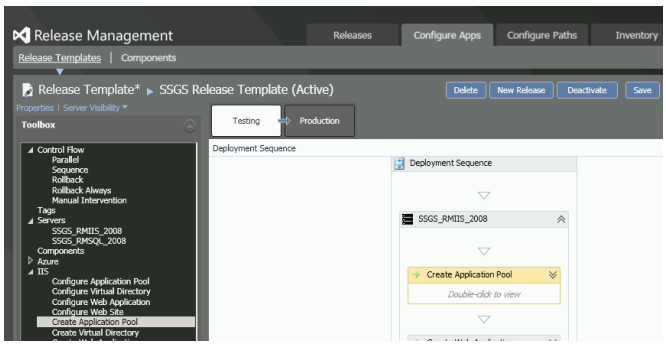
Configure IIS Server Deployment

As a first step, drag and drop IIS Server on the designer surface of the workflow.

We need to do three activities on this server. Copy the output of build to a folder on IIS server, create an application pool for the deployed application and then create an application that has the physical location which is the copied folder and the application pool. We will do the first step using a tool that is meant for XCOPY. For that, right click on Components in the tool box and click Add. From the drop down selection box, select the tool for XCOPY. In the properties of that tool, as a source, provide the UNC path (share) of the shared location where build was dropped. For example, in my case it was \\SSGS-TFS2013\BuildDrops\TicketM, the last being the name of the selected build definition. Close the XCOPY tool. Drag and drop that from the component on to the IIS Server in the designer. For the parameter installation path, provide the path of folder where you want the built files to be dropped for example C:\MyApps\HR App.

Now expand the node of IIS in the tool box. From there, drag and drop the activity of *Create Application Pool* below the XCOPY tool on the IIS Server in the designer surface. For this activity, provide the Name of the Application Pool as HR App Pool. Provide the rest of the properties as necessary for your application.

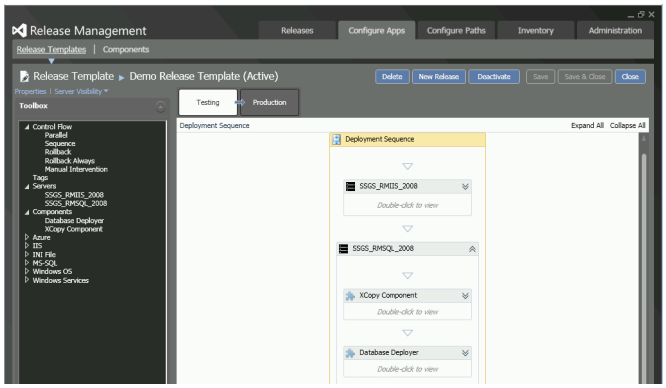
Next activity is the *Create Web Application* again from the IIS group. Drag and drop that below the Create App Pool activity. Set properties like physical folder, app pool that were created in earlier steps.



Configure Database Deployment

Database deployment can be done with the help of a component that can use SQL Server Database Tools (SSDT). We should have a SSDT project in the solution that we have built. Support for SSDT is built in Visual Studio 2013. We have to add a Database Project in the solution and add the scripts for creation of database, stored procedures and other parts of the database. When compiled, the SSDT creates a DACPAC – a database installation and update package. That becomes the part of the build drop.

Drag and drop the database server from the servers list in toolbox to the designer surface. Position it below IIS that was used in earlier steps. Now add a component for DACPAC under the components node. In that component, set the properties of Database server name, database name and name of the DACPAC package. Now drag and drop that component on the database server on the designer.



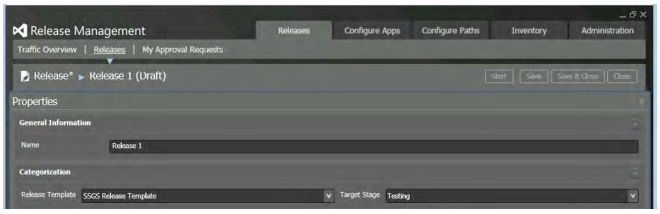
This completes the deployment workflow design for the Testing stage and now we can save the release template that was created.

STARTING A RELEASE

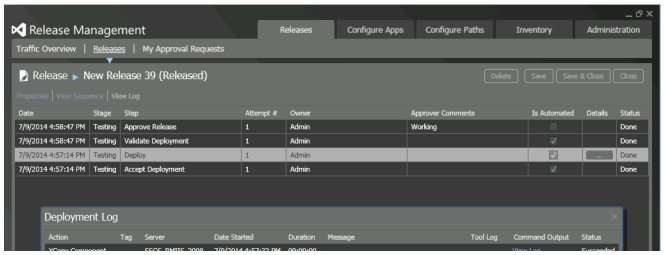
TRIGGER RELEASE FROM RM CLIENT

To start a release, you will select the activity of Releases. Click

the 'New' button to create a new release. Provide a name to the release. Select the release template that you had created earlier. Set the target stage as Testing since you have not yet configured the Production stage. You can start the release from here after saving or you can click Save and Close and then from the list of releases which includes recently created release, you can select and start it.



Once you click on the Start button, you will have to provide path of the build with the name of the latest build which is to be deployed. After providing that information, you can now start the run of the new release.



The RM Client will show you the status of various steps that are being run on the servers configured for that release. It will do the deployment on both the servers and validate the deployment automatically.

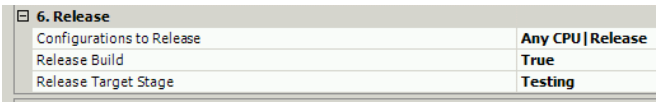
Since we had configured the Testing stage to have approval at the end, the release status will come to the activity of Approve Release and wait. After testing is over, the test lead who is configured to give approval, can open the RM Client and select the Approved button to approve the release.

We have now configured Continuous Integration with the help of team build and then configured continuous deployment with the help of release management feature of TFS 2013. Let us now integrate both of these so that automated deployment starts as soon as the build is over. To do that, we need to configure the build to have appropriate activity that triggers the release.

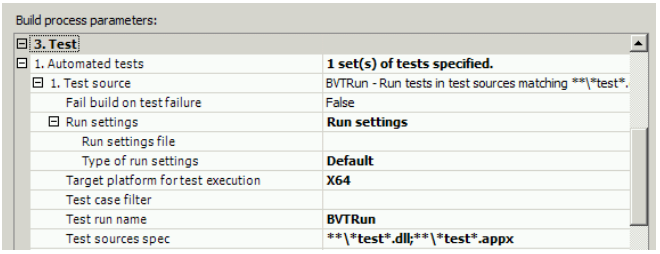
TRIGGER THE RELEASE FROM BUILD

To trigger the release from Build, we need to use a special build process template that has the activities related to triggering the build. That process template is available when you install Release Management Client on TFS Build Server (build service restart is needed). It can also be downloaded from http://blogs.msdn.com/b/visualstudioalm/archive/2013/12/09/how-to-modify-the-build-process-template-to-use-the-option-trigger-release-from-build.aspx. After that, you will need to put it under the Build Templates folder of your source and check-in.

Once the special build process template is in place, you can start the wizard for creating new build definition. Let the trigger be manual but if you want to make it part of the Continuous Integration, then you can change the trigger to that. In the build defaults, provide the shared path like \\SSGS-TFS2013\BuildDrops. If you do not have a share like that, then you can create a folder and share it for everyone. You should give write permission to that share for the user that is the identity of build service. Under the process node, select the template ReleaseTfvcTemplate.12.xaml. It adds some of the parameters under the heading of Release. By default the parameter *Release Build* is set to *False*. Change that to *True*. Set Configuration to Release as Any CPU and Release Target Stage as Testing.



Under the Test section provide the Test Run Name as *BVTRun* Save the build definition.



Open the RM Client and Create a new Release Template that is exactly same as the one created earlier. In the properties, select the checkbox of "Can Trigger a Release from a Build". In each of the components used under the source tab, select the radio button of Builds with application and in the textbox of the Path to package, enter \ character.

Now you can trigger the build and watch the build succeed. If you go to the Release Management Client after the build is successfully over, you will find that a new release has taken place.

We may also need to run some automated tests as part of the release. For that we need to configure release environment that has automated tests that run on the lab environment. So it is actually a combination of lab and release environment that helped us to run automated tests as part of the release.

AUTOMATED TESTING AS PART OF THE RELEASE

Automated testing can be controlled from Lab Management only. For this, it is necessary that we have a test case with associated automation that runs in an environment in Lab Management. The same machines that is used to create Release Management environment, can be used to create the required Lab Environment.

CREATE LAB ENVIRONMENT FOR AUTOMATED TESTING

For this, we will create a Standard Environment containing the same machines that were included in the Testing environment. Name that environment as *RMTesting*. Before the environment is created, ensure that Test Controller is installed either on TFS or other machine in the network and is configured to connect to the same TPC that we are using in our lab. While adding the machines to Lab Environment, if those machines do not have Test Agent installed, then it will be installed.

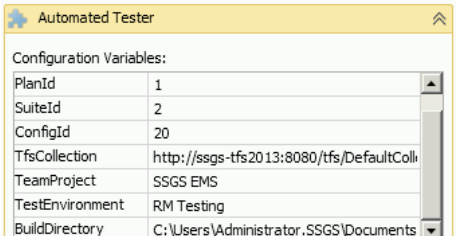
CREATE AND CONFIGURE TEST CASE TO RUN AUTOMATICALLY

Open Visual Studio 2013 and then open Team Explorer – Source Control tab. Create a new Unit Test for that method by adding a new Test project to the same solution. Check-in both the projects.

In the Testing Center of the MTM 2013, add a Test Plan if it does not exist. Create a new Test Suite named BVTests under that test plan. Open the properties page of the selected test plan and assign the created environment to that test plan.
Create a test case from Organize Activity – Test Case Manager. There is no need to add steps to the test case as we will run this test case automatically. Click on the tab of *Associate Automation*. Click the button to select automated test to associate. From the list presented to us, select the unit test that you had created in earlier steps. Save test case. Add that test case to a test suite named BVTests.

ADD COMPONENT TO RELEASE MANAGEMENT PLAN TO RUN AUTOMATED TEST

In the RM Client, open the Release Template that was used in the earlier exercise to trigger the release from build. Add a new component and Name it as Automated Tester. Under the Deployment tab of that component, select the tool MTM Automated Tests Manager. Let the other parameters as they are. Save and close the component. Drag and drop that component from toolbox on the designer. It should be on the IIS server below all the deployment activities. Open that component and set the properties as shown below. Test Run name is what you have provided in the build definition.



Now you can trigger the build again. After the build, the release is triggered. During the release after the deployment is over, automated test will be run and the run results will be available in MTM 2013.

SUMMARY

Today, teams are becoming more and more agile. It is necessary for them to be agile with the demonstrable application at any time. To support that, the practices of Continuous Integration and Continuous Deployment have become very important factors in all the practices. In this article, we have seen how Team Foundation Server 2013 with the component of Release Management helps us to implement Continuous Integration and Continuous Deployment ■

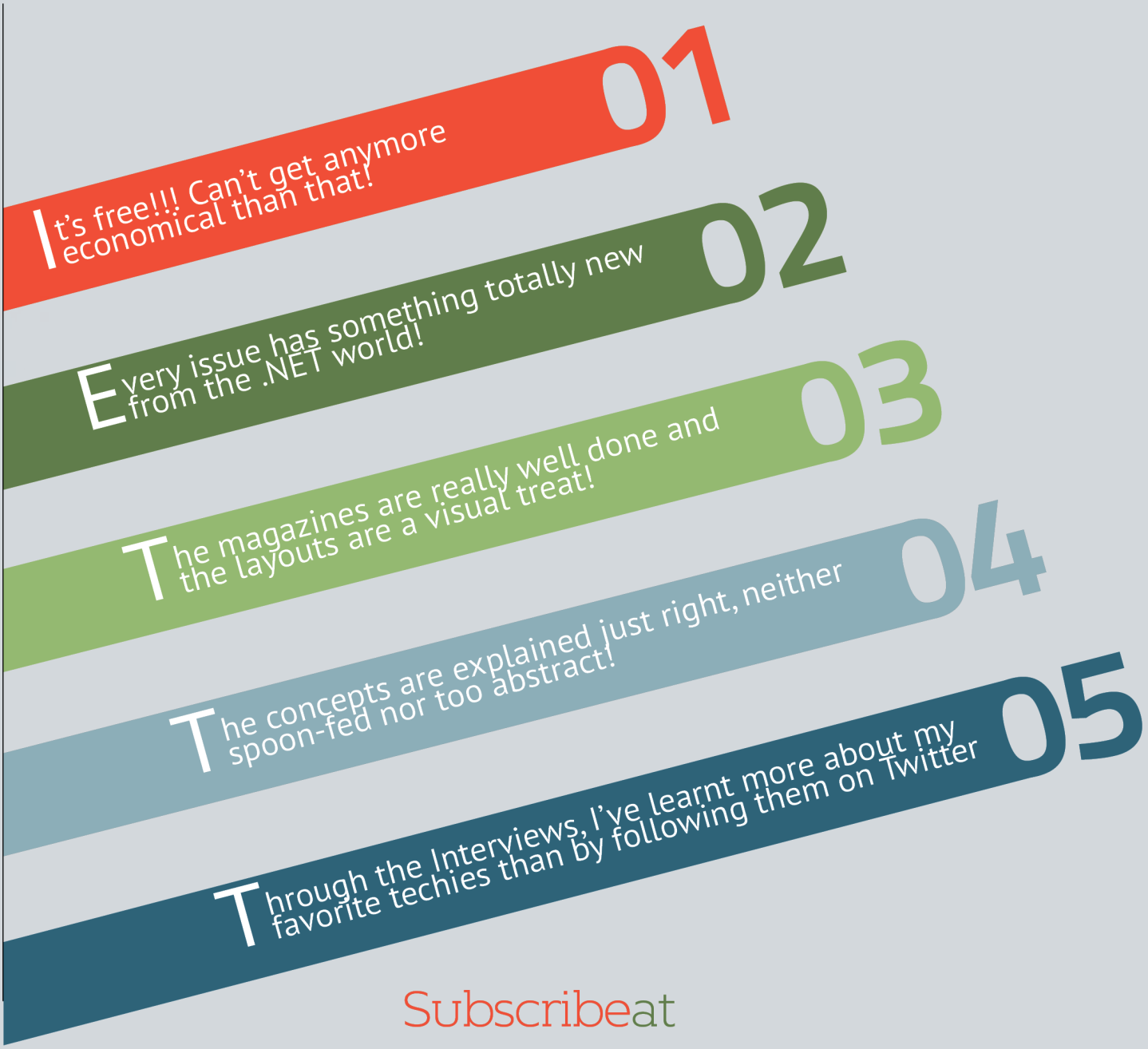
AUTHOR PROFILE

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 bit.ly/Ns9TNU

5 REASONS YOU CAN GIVE YOUR FRIENDS TO GET THEM TO SUBSCRIBE TO THE

DNC MAGAZINE

(IF YOU HAVEN'T ALREADY)



Subscribeat

www.dotnetcurry.com/magazine

HELLO XAMARIN.FORMS!

Native iOS App

Native Android app

Native Windows Phone app

Shared C# User Interface Code

Shared C# App Logic

Xamarin.Forms is a simple yet powerful UI toolkit that gives an abstraction layer for creating mobile native user interfaces for iOS, Android and Windows Phone. It gives you a rich API in C# and lets you share as much code as possible across all the three platforms. Sharing code in the Xamarin platform is super easy with *Portable Class Libraries* or simply using the sweet *Shared Projects*. Until the Xamarin.Forms release, you could only share your models, business logic, data access layer and any other code that's platform independent. In fact, this is by design and is a deliberate effort to give developers the maximum advantage of using platform specific features, without compromising on sharing business logic.

Having said that, not every app needs to have a platform level customization. For e.g. for enterprises who want to build an app to fill up their backend databases - all that they need is just a few forms in their app. Building these forms three times (for iOS, Android and Windows Phone) is definitely not a great experience. So there was a high demand for a UI abstraction layer that gives a simple API to write one piece of code and magically render respective native interfaces on every platform.

When we set out to build Xamarin.Forms, we knew it was never easy because every platform differs in subtle ways. For e.g., Navigation - iPhones and iPads have just one physical button that is "Home" and all the back navigations were done by a button on the top left corner within the app. However Android and Windows had *hardware* level Back buttons. Even some of the controls were rendered differently on each platform for e.g. Tabs in iOS had the Tab buttons at the bottom, Android had them below the top navigation bar and Windows Phone doesn't have it at all - it's a Pivot control there.

So it took us nearly 18 months to build an API that cleanly abstracts away the platform divergence and renders appropriate platform controls. Xamarin.Forms is more than just the API. It's also the patterns library that helps developers write much cleaner and maintainable code. You can leverage your Model-View-ViewModel skills with data binding, which we think is the most powerful feature of this release.

In this article, I will take you through the steps involved in building your first fully loaded Xamarin.Forms app. You will see how you can maintain a single code base across iOS, Android and Windows Phone; leverage your XAML data binding skills, learn to customize a native control and write platform specific code (you do have it when you want it!).

I understand, in the previous issue of DNC magazine, Filip Ekberg wrote a fantastic article titled Introducing Xamarin - A REST API based HackerNews application that works on iOS, Android and Windows Phone . I highly recommend you to read that article if you haven't.

BUILDING A VEHICLE SERVICE REMINDER

I often forget when was the last time I serviced my car and when its next service is due, so I built an app that reminds me when my car is due for service. I'm going to take you through a step-by-step approach of building this app using XAML and the MVVM pattern. The code is all out there in GitHub for free. Feel free to fork and add your favorite feature!

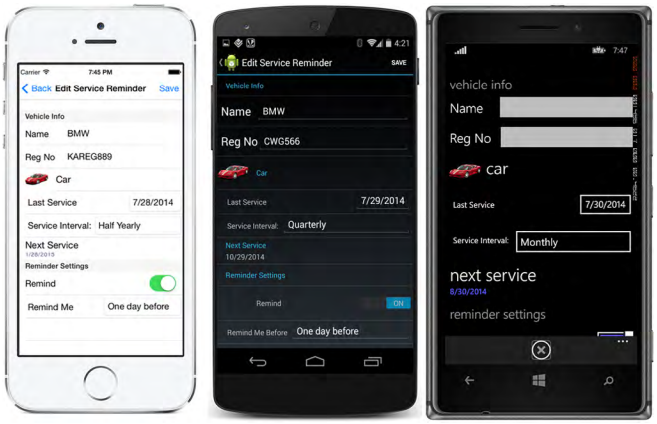
BEFORE WE START

What we need is –

- Two Screens
  - o A form that can input Name, Registration Number, Vehicle Type(Car, bike etc), Last Service Date, Next Service Date and a Reminder Date
  - o A list that shows all the Vehicles and they due dates (Typically a Home Screen)

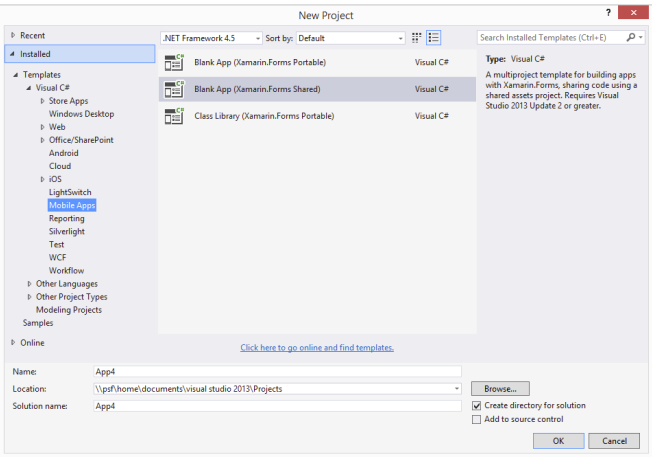
- SQLite Data Access layer

- Reminder Service (Differs in all platforms for e.g. `UILocalNotification` in iOS, `Microsoft.Phone.Scheduler.Reminder` in Windows Phone and `AlarmManager` in Android.)



CREATING A NEW PROJECT

When you install Xamarin, you get a Mobile Apps project template right inside Visual Studio. To create a new Xamarin.Forms project, File -> New -> Project -> Mobile Apps and choose Blank App (Xamarin.Forms Shared)



PCL VS SHARED PROJECT, WHAT'S BEST?

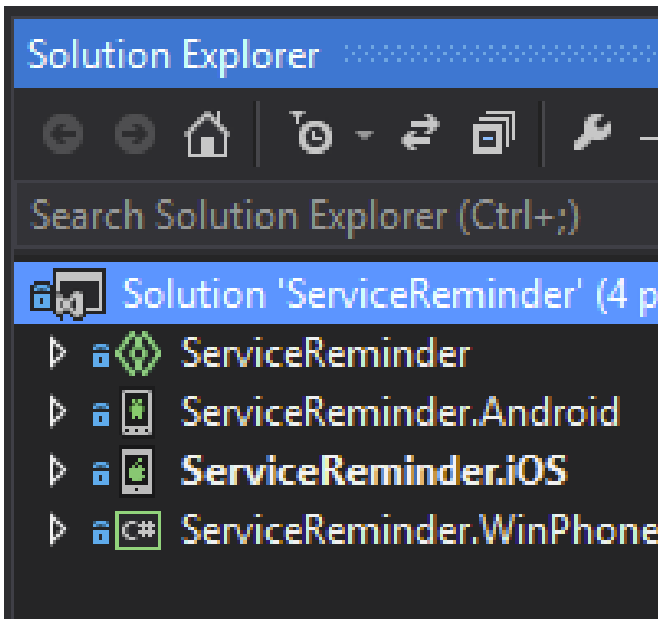
Choosing between a Shared Project or Portable Class Library (PCL) totally depends on how you want to architect your application. Both have their own advantages and disadvantages. For e.g. Shared Projects is nothing but an arbitrary project type that has files linked to referencing projects. They can be thought of as files within the referencing app, than a project that gives you a dll or exe. You're free to make use of conditional compiling statements that gives you flexibility in adding platform specific code within this project giving you space to share more code. Yes! Sometimes your code will look ugly and as your code grows, maintaining this could be harder.

PCL on the other hand is simply a compiled library that will be referenced on the target platform apps just like any another library. Since conditional compiling is not supported, you may have to use some patterns like Provider Pattern to better maintain the code. PCL works great if you are a third party library provider who ships updates often.

As said above, it's totally your architectural decision to choose between the two – you know what's best for your app.

PROJECT STRUCTURE

Xamarin.Forms automatically creates four projects in Visual Studio –



- Shared Project – Where all the shared logic resides
- Android Project – Android specific platform code
- iOS Project – iOS specific platform code
- Windows Phone Project – WP specific platform code

Ideally, you don't have to write anything inside the platform specific code, because Xamarin.Forms will wire things up for you!

CREATING THE USER INTERFACE IN XAML

There are two approaches to create user interfaces in Xamarin.Forms. The first one is to create UI views entirely with source code using the Xamarin.Forms API. The other option available is to use Extensible Application Markup Language (XAML), which is the approach I've taken to build this demo.

Please remember XAML is a markup language that follows xml syntax and the controls that you will be referencing here are part of the `Xamarin.Forms.Controls` namespace unlike the `System.Windows` in WPF. Though you will find a lot of similarities in the code, they aren't the same. For e.g. In Xamarin.Forms, You have the `<StackLayout>` instead of `<StackPanel>`. However they do the same thing – Layout controls in a stack.

A PAGE

A page is the topmost visual element that occupies the entire screen and has just a single child. You can choose from five different pages –

- ContentPage – A single view often a container to Visual Elements

- MasterDetailsPage – A page that manages two panes

- NavigationPage – A page that stacks other content pages – usually used for Navigation

- TabbedPage – A page that allows navigation using Tabs

- CarouselPage – A page that supports swipe gestures between subpages (e.g. gallery)

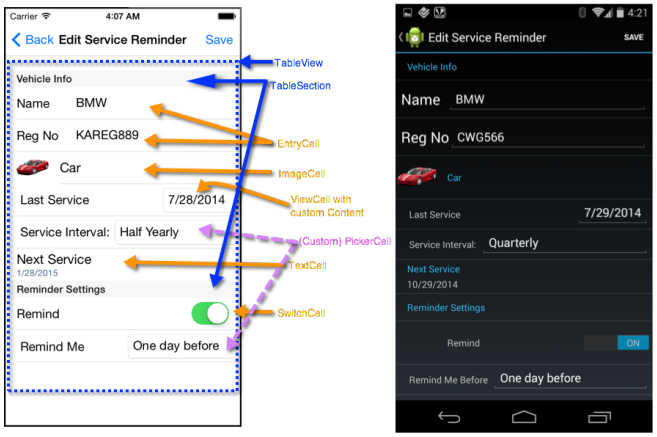
Now, let's start writing our app starting with Pages.

EDITREMINDERPAGE.XAML

EditReminderPage is a ContentPage that has the XAML for creating visual elements for Name, Registration Number, Vehicle Type(Car, bike etc), Last Service Date, Next Service Date, Reminder Date and Save implementation. These controls can be placed on the screen using any of the seven layouts or Views provided by the Xamarin.Forms API.

For this page, I used a TableView that holds the rows of cells. The advantage of using TableView is that it gives you Human Interface Guidelines Compliant user interfaces and they are extremely flexible in terms of customizing each cell.

The following screenshot shows what cells are used and how they are visually rendered with the appropriate controls in iOS and Android.



Let's look at it one by one -

TABLESECTION

TableSection logically groups elements on the screen. It provides an appropriate title to every group.

```
<TableView>
  <TableView.Root>
    <TableSection Title="Vehicle Info">
      <EntryCell/>
        <TableSection Title="Reminder Settings">
      <EntryCell/>
    </TableSection>
  </TableView.Root>
</TableView>
```

ENTRYCELL, TEXTCELL, IMAGECELL & SWITCHCELL

An *EntryCell* provides a label and a single line text entry field. In our app, this is best suited for the Vehicle Name and Registration No.

```
<EntryCell Label="Name  " Text="{Binding Name,
Mode=TwoWay}"></EntryCell>

<EntryCell Label="Reg No" Text="{Binding
RegistrationNo, Mode=TwoWay}"></EntryCell>
```

A *TextCell* consists of a Primary and a Secondary Text. They're set using Text and Detail properties.

```
<TextCell Text="Next Service"  Detail="{Binding
NextServiceDate}" ></TextCell>
```

An *ImageCell* is simply a TextCell that includes an image.

```
<ImageCell ImageSource="{Binding VehiclePhoto}"
Text="{Binding VehicleType, Mode=TwoWay}"
Command="{Binding ChooseVehichleTypeCommand}" ></
ImageCell>
```

A *SwitchCell* is a cell with a label and an on/off switch.

```
<SwitchCell Text="Remind" On="{Binding IsReminder,
Mode=TwoWay}" ></SwitchCell>
```

Notice the Data Binding syntax above – Yes, you guessed that right! You can leverage data bindings in Xamarin.Forms apps.

USE VIEWCELL TO CUSTOMIZE A CELL

Cells can be customized completely by using ViewCell. ViewCell in this case is used to incorporate a DatePicker and a Label. Here's how they look on all three platforms!


```
<ViewCell>
  <StackLayout Padding="20,5,5,5"
  HorizontalOptions="Fill" Orientation="Horizontal">
    <Label x:Name="pickerText" Text="Last Service"
    HorizontalOptions="FillAndExpand" YAlign="Center"/>
    <DatePicker x:Name="datePicker"  Date="{Binding
    LastServiceDate, Mode=TwoWay}" Format="d"
    HorizontalOptions="Fill" >
    </DatePicker>
  </StackLayout>
</ViewCell>
```

REUSING CUSTOM CELLS (PICKERCELLVIEW.XAML)

In my app, I use a Picker for choosing the *Service Interval* and *Remind Me elements*. I take a different approach here to show you how you can customize the picker to completely act on a ViewModel that is provided. I first create a `ContentView` in XAML and name it `PickerCellView.xaml`

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentView xmlns="http://xamarin.com/schemas/2014/
forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="ServiceReminder.Cells.PickerCellView">
  <StackLayout Padding="20,5,5,5"
  HorizontalOptions="Fill" Orientation="Horizontal">
    <Label x:Name="pickerText" Text="{Binding
    PickerText}" HorizontalOptions="Fill"
    YAlign="Center"/>
    <Picker x:Name="picker"  Title="{Binding
    PickerTitle}" HorizontalOptions="FillAndExpand">
    </Picker>
  </StackLayout>
</ContentView>
```

Next, I create a PickerCell<T> that inherits from ViewCell where T is a ViewModel that implements the IPickerCellViewModel. Finally, set the View of the PickerCell to the Picker.

```
public class PickerCell<T> : ViewCell where T :
IPickerCellViewModel
{
  public static readonly BindableProperty
  PickerCellViewModelProperty =
  BindableProperty.Create<PickerCell<T>, T>(p=>
 p.PickerCellViewModel, Activator.CreateInstance<T>());

  public T PickerCellViewModel
  {
    get { return (T)
    GetValue(PickerCellViewModelProperty); }
    set { SetValue(PickerCellViewModelProperty, value);}
  }

  PickerCellView vw;
  public PickerCell()
  {
    var vw = new PickerCellView();
    vw.BindingContext = PickerCellViewModel;
    View = vw;
  }
}
```

A Note on BindingContext & BindableProperties

Bindings are used most often to connect the visuals of a program with an underlying data model, usually in a realization of the MVVM (Model-View-ViewModel) application architecture. A BindingContext is just like DataContext in WPF and usually set to your ViewModels. BindableProperties are just like DependencyProperties in WPF. Create them when you want the properties to be participating in Data Bindings. In this case, we are setting the ViewModel as the BindableProperty and assigning it to the BindingContext of the control so that any changes to the property are reflected in the UI and vice versa.

Finally, these custom cells are used in the TableView with appropriate ViewModels

```
<cells:PickerCell x:TypeArguments=
”vm:ServiceIntervalPickerCellViewModel”/>
<cells:PickerCell
x:TypeArguments=”vm:RemindMePickerCellViewModel”/>
```

KEEP ONPLATFORM<T> HANDY

Sometimes you have the need to write platform specific code in XAML. In my case, I wanted to set a padding of 20 in the ContentPage in iOS so as to avoid the overlapping of the status bar. Since the 20 units on top is only required in case of iOS, I can make use of OnPlatform<T> to set that.

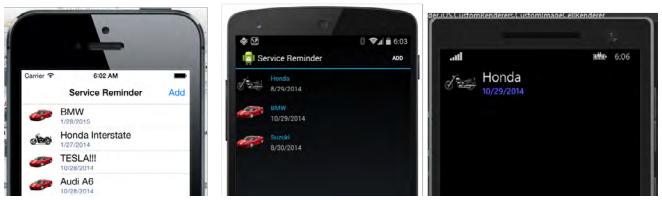
```
<ContentPage.Padding>
  <OnPlatform x:TypeArguments=”Thickness”>
    <OnPlatform.iOS>
      0, 20, 0, 0
    </OnPlatform.iOS>
    <OnPlatform.Android>
      0, 0, 0, 0
    </OnPlatform.Android>
    <OnPlatform.WinPhone>
      0, 0, 0, 0
    </OnPlatform.WinPhone>
  </OnPlatform>
</ContentPage.Padding>
```

HOMEPAGE.XAML

It’s time to create a homepage that displays the list of created Service Reminder Items.

USE A LISTVIEW FOR REPEATING DATA

A ListView is an ItemView that displays a collection of data as a vertical list.




```
<ListView x:Name=”listView”>
 <ListView.ItemTemplate>
  <DataTemplate>
   <ImageCell ImageSource=”{BindingVehiclePhoto}”
   Text=”{Binding Name}” Detail=”{Binding
   Path=NextServiceDate}”></ImageCell>
  </DataTemplate>
 </ListView.ItemTemplate>
</ListView>
```

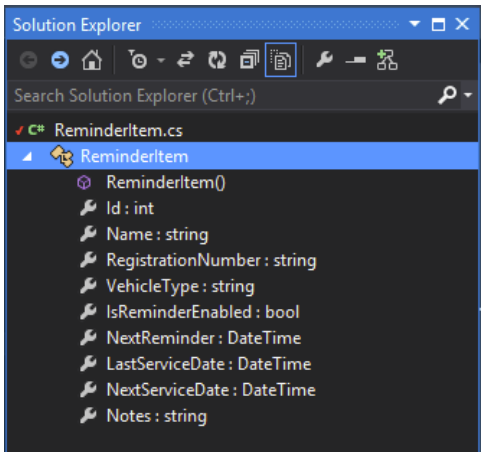
HANDLING DATA (REMINDERITEMDATABASE.CS)

Xamarin.Forms apps can easily access local SQLite databases, using compiler directives with Shared Projects or using PCL. Because the Shared Project code is treated as part of each individual application project, the ReminderItemDatabase class can take advantage of compiler directives to perform platform-specific operations while sharing the remainder of the code in the class.

```
static string DatabasePath
{
  get
  {
    var sqliteFilename = “ServiceReminder.db3”;
    #if __IOS__
    string documentsPath = Environment.
    GetFolderPath(Environment.SpecialFolder.Personal);
    // Documents folder
    string libraryPath = Path.Combine(documentsPath,
    “..”, “Library”); // Library folder
    var path = Path.Combine(libraryPath, sqliteFilename);
    #else
```

```
    #if __ANDROID__
    string documentsPath = Environment.GetFolderPath
    (Environment.SpecialFolder.Personal); // Documents
    folder
    var path = Path.Combine(documentsPath,
    sqliteFilename);
    #else
    // WinPhone
    var path = Path.Combine(ApplicationData.Current.
    LocalFolder.Path, sqliteFilename);;
    #endif
    #endif
    return path;
  }
}
```

MODEL (REMINDERITEM.CS)



Add some helper methods to save and retrieve our models.

```
static object locker = new object();
  static SQLiteConnection database = new SQ
  LiteConnection(databasePath: DatabasePath);

  public ReminderItemDatabase()
  {
    // create the tables
    database.CreateTable<ReminderItem>();
  }

  public IEnumerable<ReminderItem> GetItems()
  {
    lock (locker)
    {
      return (from i in database.Table<ReminderItem>()
      select i).ToList();
    }
  }

  public ReminderItem GetItem(int id)
  {
    lock (locker)
    {
      return database.Table<ReminderItem>().
      FirstOrDefault(x => x.Id == id);
    }
  }

  public int SaveItem(ReminderItem item)
  {
    lock (locker)
    {
      if (item.Id != 0)
      {
        database.Update(item);
        return item.Id;
      }
      else
      {
        return database.Insert(item);
      }
    }
  }

  public int DeleteItem(int id)
  {
    lock (locker)
    {
      return database.Delete<ReminderItem>(id);
    }
  }
```

SQLITE.CS

To use SQLite with a Xamarin.Forms Shared Project, simply add the SQLite.Net ORM source to the project. (bit.ly/dncm14-sqllite)

Please note: By default, Windows Phone does not include the SQLite database engine, so you need to ship it with your application. Download the Pre-compiled binaries for Windows

Phone 8 from here http://www.sqlite.org/download.html#wp8 and install them.

ADD A REMINDER (PLATFORM SPECIFIC CODE)

Until now, whatever code I wrote is in the Shared Project and the appropriate native interfaces were rendered on all three platforms – iOS, Android and Windows Phone. Now let me show you how to write some platform specific code.

The Service Reminder app is incomplete without adding a reminder service – i.e. when the user saves the ReminderItem, the app should remind the users at the set date and time. The easiest way to implement this is by using the platform specific Reminder Services. In iOS we can use UILocalNotification, in Android it's the Alarm Manager and in Windows Phone it's the Microsoft.Phone.Scheduler.Reminder.

Xamarin.Forms includes a DependencyService to let shared code easily resolve Interfaces to platform-specific implementations, allowing you to access features of the iOS, Android and Windows Phone SDKs from your Shared Project.

There are three parts to a DependencyService implementation:

- **Interface** - An Interface in the shared code that defines the functionality

- **Registration** - An implementation of the Interface in each platform-specific application project

- **Location** – Call DependencyService.Get<> in shared code to get a platform-specific instance of the Interface at run time, thereby allowing the shared code to access the underlying platform.

Interface (IReminderService.cs)

```
public interface IReminderService
{
  void Remind(DateTime dateTime, string title, string
  message);
}
```

Registration and Platform Implementation (iOSReminderService.cs)

This code is written in the iOS project and it requires you to specify the *[assembly] attribute*, which does the registration of your implementation.

```
[assembly: Xamarin.Forms.
Dependency(typeof(iOSReminderService))]
namespace ServiceReminder.iOS.ReminderService
{
  public class iOSReminderService : IReminderService
  {
    public void Remind(DateTime dateTime, string title,
    string message)
    {
      var notification = new UILocalNotification();

      //---- set the fire date (the date time in which
      it will fire)
      notification.FireDate =dateTime;

      //---- configure the alert stuff
      notification.AlertAction = title;
      notification.AlertBody = message;

      //---- modify the badge
      notification.ApplicationIconBadgeNumber = 1;

      //---- set the sound to be the default sound
      notification.SoundName = UILocalNotification.
      DefaultSoundName;

      //---- schedule it
      UIApplication.SharedApplication.
      ScheduleLocalNotification(notification);
    }
  }
}
```

(WPReminderService.cs)

Just like iOS, WP implementation has to be written in the WP project.

```
[assembly: Xamarin.Forms.
Dependency(typeof(WPReminderService))]
```

```
namespace ServiceReminder.WinPhone.ReminderService
{
  public class WPReminderService : IReminderService
  {
    public void Remind(DateTime dateTime, string title,
    string message)
    {
      string param1Value = title;
      string param2Value = message;
      string queryString = "";
      if (param1Value != "" && param2Value != "")
      {
        queryString = "?param1=" + param1Value +
        "&param2=" + param2Value;
      }
      else if (param1Value != "" || param2Value != "")
      {
        queryString = (param1Value != null) ?
        "?param1=" + param1Value : "?param2="
        + param2Value;
      }
      Microsoft.Phone.Scheduler.Reminder reminder =
      new Microsoft.Phone.Scheduler.
      Reminder("ServiceReminder");
      reminder.Title = title;
      reminder.Content = message;
      reminder.BeginTime = dateTime;
      reminder.ExpirationTime = dateTime.AddDays(1);
      reminder.RecurrenceType = RecurrenceInterval.
      None;
      reminder.NavigationUri = new Uri("/MainPage.xaml"
      + queryString, UriKind.Relative);
      ;

      // Register the reminder with the system.
      ScheduledActionService.Add(reminder);

    }
  }
}
```

Android – How about you Fork the project and implement it. Ping me on twitter at @nishanil with the link and if I like your solution, I will send you a Xamarin goodie. * Hint: use Alarm Manager

CONCLUSION

Xamarin.Forms makes developing mobile applications on all three major platforms a breeze. It's a nice abstraction layer packaged with a patterns library that makes it super easy for any C# developer to write their next big app on iOS, Android and Windows Phone within no time. Xamarin makes it easy, fast and fun for you! Get your hands dirty now! ■

Download the entire source code from our GitHub Repository at bit.ly/dncm14-xamarinforms

AUTHOR PROFILE

Nish is a C# fanatic and has been writing softwares for web, windows & mobile platforms for almost a decade now. He is Xamarin's first Developer Evangelist for India and a Microsoft MVP in Visual C# . He is passionate about spreading C# awesomeness and Xamarin love by helping developers build 100% native mobile applications for iOS, Android, Windows Phone & Windows platforms. Follow his adventures on twitter @nishanil

THE ABSOLUTELY AWESOME

.NET INTERVIEW BOOK

SUPROTIM AGARWAL

PRAVIN DABADE

REAL-TIME DATA VISUALIZATION

USING D3 AND ASP.NET SIGNALR

The wondrous web is evolving! It constantly changes and sometimes these changes and standards can be overwhelming enough for any web developer to ignore. In its modern incarnation, one of the hottest things that the web has delivered is the support for real-time communication. In this mode of communication, the server pushes new data to the client as soon as it is available. This communication could be in the form of a live chat, notification, in exciting apps like Twitter, Facebook and even real-time financial data. The client can then receive this data and project it to the end user the way he/she wants.

> *I decided that if I could paint that flower in a huge scale, you could not ignore its beauty.*
>
> – Georgia O' Keefe

Another area where the web has taken big leaps is in Data Visualization. HTML5 Canvas and Scalable Vector Graphics (SVG) are both web technologies that enable us to create rich graphics in the browser, without the need of a plugin like Flash or Silverlight. One of the several useful cases where we need graphics in the browser is to draw some interactive charts. These charts may range from simple line charts to very sophisticated 3D graphic charts.

By combining Real-time communication with Data Visualization, we can build some real cool sites that would impress our customers. In this article, we will see how to leverage ASP.NET SignalR (real-time communication) and D3 (JavaScript charting library) to build real-time charts in browsers.

A BRIEF NOTE ON SIGNALR AND D3

SIGNALR

There are several techniques to implement real-time communication in web applications like Interval Polling, Long Polling, Server Sent Events, Web Sockets etc. HTML 5 Web Sockets is one of the most popular techniques amongst them. In order for Web Sockets to work, they should be supported on both the client as well as the server. If any of them doesn't support Web sockets, the communication can't happen. In

such cases, we need a polyfill that includes some fallback mechanisms. ASP.NET SignalR provides this feature for free. If we use SignalR, we don't need to manually detect the supported technologies on the running platforms. SignalR abstracts all the plumbing logic needed to detect the feature supported by both communication ends and uses the best possible feature. SignalR checks and uses one of the following techniques in the order listed below:

- Web Sockets: A feature of HTML5 for real-time communication using a special protocol 'ws'

- Server Sent Events: A feature of HTML5 for real-time communication over HTTP and HTTPS

- Forever Frames: Adds a hidden iFrame on the page and manages communication using the frame

- Long Polling: Client keeps on invoking the server after certain interval to check if the server has any data to be pushed to the client

To learn more about SignalR, you can refer to the following sources:

- Official ASP.NET Site
- SignalR articles on Dotnet Curry

D3.JS

As already mentioned, most of the modern browsers have good support for graphics because of features like: Canvas and SVG. JavaScript API on the browser enables us to talk to these components through code, and manipulate them as needed. When we start learning these new features, the APIs seem very nice initially, but we start getting tired of them once we get into rendering graphics beyond simple drawing. This is where we need an abstraction that wraps all default browser APIs and makes it easier to get our work done.

D3 is an abstraction over SVG. D3 makes it easier to work with SVG and provides APIs to create very rich graphics. To learn more about D3, please check the official site.

STOCK CHARTS APPLICATION USING SIGNALR AND D3

We are going to create a Stock Charts application. This is a simple application that maintains details of stocks of some companies and then displays a simple line chart to the users that shows the change in stock value of a company. Values of stocks keep changing after every 10 seconds. A real-time notification is sent to all the clients with the latest value of the stock data. Charts on the client browsers are updated as soon as they receive new data from the server.

Open Visual Studio 2013 and create a new Empty Web Project named StockCharts-SignalR-D3. In the project, install the following NuGet packages:

- EntityFramework
- Microsoft.AspNet.SignalR
- Microsoft.AspNet.WebApi
- Bootstrap
- D3

Now that we have all the required hooks, let's start building the application.

BUILDING SERVER SIDE COMPONENTS

SETTING UP DATABASE USING ENTITY FRAMEWORK

On the server side, we need Entity Framework to interact with SQL Server and Web API, SignalR to serve data to the clients. In the database, we need two tables: Companies and StockCosts. We will use Entity Framework Code First to create our database.

To get started, let's define the entity classes:

```
public class Company
{
    [Key]
    public int CompanyId { get; set; }
    public string CompanyName { get; set; }
}

public class StockCost
{
```

```
    public int Id { get; set; }
    public int CompanyId { get; set; }
    public double Cost { get; set; }
    public DateTime Time { get; set; }
    public Company Company;
}
```

We need a *DbContext* class that defines *DbSets* for the above classes and tables are generated automatically for these classes when the application is executed or when the migrations are applied. Following is the context class:

```
public class StocksContext: DbContext
{
    public StocksContext()
        : base("stocksConn")
    {

    }
    public DbSet<Company> Companies { get; set; }
    public DbSet<StockCost> StockCosts { get; set; }
}
```

The parameter passed into the base class (DbContext) in the above snippet is the name of the connection string. Let's add a connection string with the name specified to the Web.config file:

```
<add name="stocksConn" connectionString="Data Source=.;
Initial Catalog=StocksDb; Integrated Security=true;"
providerName="System.Data.SqlClient" />
```

I am using SQL Server as my data source. You can change it to SQL CE, local DB or any source of your choice.

When a database with the above tables gets created, the tables don't contain any data. Entity Framework migrations provide an easier way to seed the database with some default data. To enable migrations, open Package Manager Console and type the following command:

```
>Enable-Migrations
```

The above command creates a new folder named *Migrations* and adds a class called *Configuration* to the folder. The command is smart enough to detect the DbContext class and use it to create the Configuration class.

The constructor of the Configuration class contains a statement that switches off the automatic migrations. For this application, we need to enable the option. Replace the statement as:

```
AutomaticMigrationsEnabled = true;
```

The *Seed* method is executed every time we create the database using migrations. Seed method is the right place to seed some data to the database. One important point to remember is, by default the Seed method doesn't check for existence of data. We need to check a condition and run the logic of seeding data, only when the database doesn't have any data. Following is the definition of the seed method:

```
protected override void Seed(StocksContext context)
{
  if (!context.StockCosts.Any())
  {
    context.Companies.AddRange(new List<Company>() {
      new Company(){CompanyName="Microsoft"},
      new Company(){CompanyName="Google"},
      new Company(){CompanyName="Apple"},
      new Company(){CompanyName="IBM"},
      new Company(){CompanyName="Samsung"}
  });
  var randomGenerator = new Random();

  for (int companyId = 1; companyId <= 5; companyId++)
  {
    double stockCost = 100 * companyId;

    for (int count = 0; count < 10; count++)
    {
      context.StockCosts.Add(new StockCost()
      {
        CompanyId = companyId,
        Cost = stockCost,
        Time = DateTime.Now - new TimeSpan(0, count, 0)
      });

      if (count % 2 == 0)
      {
        stockCost = stockCost + randomGenerator.
        NextDouble();
      }
      else
      {
```

```
      stockCost = stockCost - randomGenerator.
      NextDouble();
      }
    }
  }
 }
}
```

Open the Package Manager Console again and run the following command to create the database with the default data:

```
>Update-Database
```

Now if you open your SQL Server, you will see a new database created with the required tables and the tables containing default data.

We need repositories to interact with the data. The StockCostRepository class should include the following functionalities:

1. Add a new cost

2. Get last 20 costs for a specified company. If the number of records is less than 20, it should return all rows

3. Get cost of the last stock. This value will be used for further calculations

Following is the implemented class:

```
public class StockCostRepository
{
  StocksContext context;

  public StockCostRepository()
  {
    context = new StocksContext();
  }

  public List<StockCost> GetRecentCosts(int companyId)
  {
    var count = context.StockCosts.Count();
    if (count > 20)
    {
       var stockCostList = context.StockCosts.Where(sc
       => sc.CompanyId == companyId)
         .OrderByDescending(sc=>sc.Id)
         .Take(20)
         .ToList();
       stockCostList.Reverse();
       return stockCostList;
    }
    else
    {
      return context.StockCosts.Where(sc =>
      sc.CompanyId == companyId)
        .ToList();
     }
  }

  public double GetLastStockValue(int companyId)
  {
    return context.StockCosts.Where(sc =>
    sc.CompanyId == companyId)
      .OrderByDescending(sc => sc.Id)
      .First()
      .Cost;
  }

  public StockCost AddStockCost(StockCost
  newStockCost)
  {
    var addedStockCost = context.StockCosts.
    Add(newStockCost);
      context.SaveChanges();
      return addedStockCost;
  }
}
```

The company repository has the functionality to return data. Following is the implementation:

```
public class CompanyRepository
{
  StocksContext context;

  public CompanyRepository()
  {
    context = new StocksContext();
  }

  public List<Company> GetCompanies()
```

```
    {
        return context.Companies.ToList();
    }
}
```

WEB API TO SERVE COMPANIES DATA

We need a Web API endpoint to expose the list of companies. Add a new folder to the project and name it Web API. Add a new Web API controller class to this folder and name it CompaniesController. The controller needs to have just one Get method to serve the list of companies. Following is the implementation:

```
[Route("api/companies")]
public class CompaniesController : ApiController
{
  // GET api/<controller>
  public IHttpActionResult Get()
  {
    try
    {
      return Ok(new CompanyRepository().
      GetCompanies());
    }
    catch (Exception ex)
    {
      return BadRequest(ex.Message);
    }
  }
}
```

The Route attribute added to the CompaniesController is a new feature added in Web API 2. It doesn't work by default. We need to add the following statement to the Application_Start event of Global.asax to enable attribute routing:

```
GlobalConfiguration.Configure((config) =>
{
  config.MapHttpAttributeRoutes();
});
```

Note: *To learn more about the Route attribute, read* What's New in Web API 2.0.

ADDING SIGNALR TO SERVE STOCK DATA

ASP.NET SignalR has two mechanisms to talk to the clients: Hubs and Persistent connections. Persistent connections are useful to deal with the low-level communication protocols that SignalR wraps around. Hubs are one level above Persistent Connection and they provide API to communicate using data without worrying about underlying details. For the Stock charts application, we will use Hubs as the communication model. We need a hub to send the initial stock data to the clients. Create a new folder for adding functionalities of SignalR and add a new Hub named StockCostsHub to this folder. Following is the code of StockCostsHub:

```
public class StockCostsHub : Hub
{
  public void GetInitialStockPrices(int companyId)
  {
    Clients.Caller.initiateChart(StockMarket.Instance.
    PublishInitialStockPrices(companyId));
  }
}
```

As we see, all Hub classes are inherited from Microsoft.AspNet.SignalR.Hub class. The Hub classes have an inherited property, *Clients* that holds information about all the clients connected. Following are some important properties of the Clients object:

- All: A dynamic object, used to interact with all clients currently connected to the hub

- Caller: A dynamic object, used to interact with the client that invoked a hub method

- Others: A dynamic object, used to interact with the clients other than the one that invoked a hub method.

The *initiateChart* method called above has to be defined by the client. To this method, we are passing the list of stocks for the given company. We haven't built the StockMarket class yet, it is our next task.

The StockMarket is a singleton class containing all the logic to be performed. It includes the following:

1. Start the market
2. Return last 20 stock prices of a company
3. Run a timer that calls a method after every 10 seconds to calculate next set of stock prices and return this data to all clients

Add a new class to the SignalR folder and name it StockMarket. Following is the initial setup needed in this class:

```
public class StockMarket
{
  //Singleton instance created lazily
  public static readonly Lazy<StockMarket> market = new
  Lazy<StockMarket>(() => new StockMarket());

  //Flag to be used as a toggler in calculations
  public static bool toggler = true;

  //A dictionary holding list of last inserted stock
  prices, captured to ease calculation
  public static Dictionary<int, double>
  lastStockPrices;

  //To be used to prevent multiple insert calls to DB
  at the same time
  private readonly object stockInsertingLock = new
  Object();

  //Repositories and data
  CompanyRepository companyRepository;
  StockCostRepository stockCostRepository;
  List<Company> companies;

  //Timer to be used to run market operations
  automatically
  Timer _timer;

  public StockMarket()
  {
    companyRepository = new CompanyRepository();
    stockCostRepository = new StockCostRepository();
    companies = companyRepository.GetCompanies();
  }

  static StockMarket()
  {
    lastStockPrices = new Dictionary<int, double>();
  }

  public static StockMarket Instance
  {
    get
    {
      return market.Value;
    }
  }
}
```

Once we receive the first request for the stock prices, we can start the market and trigger the timer with an interval of 10 seconds. This is done by the following method:

```
public IEnumerable<StockCost>
PublishInitialStockPrices(int companyId)
{
  //Get the last 20 costs of the requested company
  using the repository
  var recentStockCosts = stockCostRepository.
  GetRecentCosts(companyId);

  //If timer is null, the market is not started
  //Following condition is true only for the first time
  when the application runs
  if (_timer==null )
  {
    _timer = new Timer(GenerateNextStockValue, null,
    10000, 10000);
  }
  return recentStockCosts;
}
```

The *GenerateStockValue* method is responsible for the following:

- Calculating the next stock value
- Insert stock value to the database using *StockCostRepository*
- Notify the new stock values to all the connected clients

It uses the lock object we created above to avoid any accidental parallel insertions to the database.

To get the list of currently connected clients to the *StockCostHub*, we need to use *GlobalHost.ConnectionManager*.

GlobalHost is a static class that provides access to the information of the Host on which SignalR runs. *ConnectionManager* provides access to the current Hubs and *PersistentConnections*. Using the generic *GetHubContext* method of the connection manager, we can get the currently connected clients and then call a client method using the dynamic Clients object. Following is the implementation of *GenerateNextStockValue* method:

```
public void GenerateNextStockValue(object state)
{
  lock (stockInsertingLock)
  {
    Random randomGenerator = new Random();
    int changeInCost = randomGenerator.Next(100, 110);
    double lastStockCost, newStockCost;
    List<StockCost> stockCosts = new List<StockCost>();

    foreach (var company in companies)
    {
      if (!lastStockPrices.TryGetValue(company.
      CompanyId, out lastStockCost))
      {
        lastStockCost = stockCostRepository.
        GetLastStockValue(company.CompanyId);
      }
      if (toggler)
      {
        newStockCost = lastStockCost + randomGenerator.
        NextDouble();
      }
      else
      {
        newStockCost = lastStockCost - randomGenerator.
        NextDouble();
      }

      var newStockAdded = stockCostRepository.
      AddStockCost(new StockCost() { Cost =
      newStockCost, Time = DateTime.Now,
      CompanyId = company.CompanyId });
      stockCosts.Add(newStockAdded);
      lastStockPrices[company.CompanyId] =
      newStockCost;
    }
    toggler = !toggler;
    GetClients().All.updateNewStockCosts(stockCosts);
  }
}

public IHubConnectionContext<dynamic> GetClients()
{
  return GlobalHost.ConnectionManager.
  GetHubContext<StockCostsHub>().Clients;
}
```

Now that all of our server side components are ready, let's build a client to consume the server components.

BUILDING CLIENT SIDE COMPONENTS

First, let's build a simple page in which we will render the chart. The page has very small amount of mark-up, later we will add a lot of JavaScript to it. Following is the mark-up inside the body tag:

```
<div class="container">
  <h3 class="page-header">
    A sample chart application demonstrating usage of
    SignalR and D3
  </h3>
  <div class="row">
    Companies: <select id="selCompanies"></select>
  </div>
  <div class="row">
    <svg id="svgStockChart" width="1000"
    height="500"></svg>
  </div>
</div>

<script src="Scripts/jquery-1.9.0.js"></script>
<script src="Scripts/jquery.signalR-2.1.0.js"></script>
<script src="SignalR/hubs"></script>
<script src="Scripts/d3.js"></script>
```

The third script library file added in the above mark-up is generated dynamically by SignalR. It contains client side proxies for all the hub methods.

INITIALIZING AND HOOKING SIGNALR PIECES

As soon as the page loads, we need to perform the following tasks:

- Get the list of companies and bind them to the select control
- Create a client hub object
- Set SignalR client callbacks
- Bind change event to the select control. The event callback has to invoke the SignalR server method to get the initial stock prices

```
var companiesSelector, hub, stockCostData, lineData,
chartObj, companies;
$(function () {
  companiesSelector = $("#selCompanies");
  setCompanies().then(function () {
    initiateSignalRConnection();
  });

  hub = $.connection.stockCostsHub;
  setSignalRCallbacks();

  ompaniesSelector.change(function () {
    hub.server.getInitialStockPrices(companiesSelector.
    val());
  });
});

function setCompanies() {
  return $.get("api/companies").then(function (data) {
    companies = data;
    var html = "";
    $.each(data, function () {
      html = html + "<option value='" + this.CompanyId
      + "'>" + this.CompanyName + "</option>";
    });
    companiesSelector.append(html);
  });
}

function setSignalRCallbacks() {
  hub.client.initiateChart = function (stocks) {
    console.log(stocks);
    stockCostData = stocks;
    transformData();
      //TODO: draw D3 chart
    };
    hub.client.updateNewStockCosts = function (stocks){
      stockCostData.push(stocks[stockCostData[0].
      CompanyId - 1]);
      stockCostData.splice(0, 1);
      transformData();
      //TODO: redraw chart
    };
}
```

The initiateSignalRConnection method called above in the callback of setCompanies establishes a real-time connection with the server. It is called in the callback as it needs data returned by the Web API. Following is the implementation:

```
function initiateSignalRConnection() {
  $.connection.hub.start().then(function () {
    hub.server.getInitialStockPrices(companies[0].
    CompanyId);
  });
}
```

The StockCost class has a property called Time, that holds date and time when the value of the cost was changed. In the line chart, it makes more sense to show the relative time when the value was changed. Also, we don't need all the properties returned by the server in the object to be plotted. Let's perform a projection on the data and get a lighter version of the data that will be easier to plot. Following transformData function does this for us:

```
function transformData() {
  lineData = [];
  stockCostData.forEach(function (sc) {
    lineData.push({ cost: sc.Cost, timeAgo: (new Date()
    - new Date(sc.Time)) / 1000 });
  });
}
```

DRAWING CHARTS USING D3

Now we have all the data ready on the client side to draw the line charts. Let's define a module function that wraps all the logic needed to draw the charts and returns an object using which we can update the chart later. Following is the skeleton of the function, and the data that the function needs:

```
function initChart() {
```

```
  var svgElement = d3.select("#svgStockChart"),
  width = 1000,
  height = 200,
  padding = 45,
  pathClass = "path";
  var xScale, yScale, xAxisGen, yAxisGen, lineFun;

  //Logic for initializing and drawing chart
  function redrawLineChart(){
    //Logic for redrawing chart
  }

  return {
    redrawChart: redrawLineChart
  }
}
```

First statement in the function selects an object of the SVG DOM element on which the chart has to be drawn. We used D3's selection API instead of jQuery, as we need a D3 object for further work. Selectors in D3 are similar to the jQuery selectors. Let's start with setting all parameters for the chart. It includes the following:

1. Setting domain and range for x-axis

2. Setting domain and range for y-axis

3. Drawing lines for x and axes with right orientation

4. Define a function that draws the line on the chart

Domain and range can be defined using D3's scale API. Range is the physical co-ordinate on which the axis will reside and domain is the minimum and maximum values to be plotted on the axis. Following snippet sets domain and range for x-axis:

```
xScale = d3.scale.linear()
           .range([padding + 5, width - padding])
           .domain([lineData[0].timeAgo,
           lineData[lineData.length - 1].timeAgo]);
```

Now we need to define an axis generator for x-axis using the scale object created above. The generator sets the position, scale and number of ticks on the axis. We will use the above scale object in the generator of x-axis.

```
xAxisGen = d3.svg.axis()
  .scale(xScale)
  .ticks(lineData.length)
  .orient("bottom");
```

Similarly, we need to define scale and generator for y-axis. Following is the code for the same:

```
yScale = d3.scale.linear()
           .range([height - padding, 10])
           .domain([d3.min(lineData, function (d) {
               return d.cost - 0.2;
           }),
            d3.max(lineData, function (d) {
                return d.cost;
            })]);

yAxisGen = d3.svg.axis()
  .scale(yScale)
  .ticks(5)
  .orient("left");
```

The last and most important parameter to be set is a function to draw the chart. This function reads values of the co-ordinates to be plotted and also defines how to join an adjacent pair of points. Following is the function that we will use to generate a linear basis chart:

```
lineFun = d3.svg.line()
            .x(function (d) {
                return xScale(d.timeAgo);
            })
            .y(function (d) {
                return yScale(d.cost);
            })
            .interpolate('basis');
```

Now, let's apply the above parameters on the SVG element to see the chart. At first, let's apply both of the axes on the element. Axes on SVG are appended as a special element, *g*. All we need to do is, append the element *g* and call the corresponding axis generator defined above.

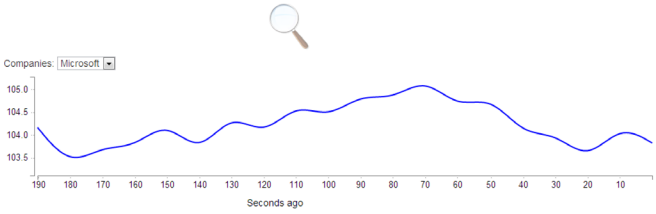
```
svgElement.append("g")
  .attr("class", "x axis")
  .attr("transform", "translate(0," + (height -
padding) + ")")
```

```
  .call(xAxisGen);

svgElement.append("g")
  .attr("class", "y axis")
  .attr("transform", "translate(" + padding + ",0)")
  .call(yAxisGen);
```

Finally, we need to append a path element using *lineFun* created above. We can set style properties like color, width and fill style using attributes on the path element.

```
svgElement.append("path")
  .attr({
    "d": lineFun(lineData),
    "stroke": "blue",
    "stroke-width": 2,
    "fill": "none",
    "class": pathClass
  });
```

Now, if you run the page, you will see a line chart similar to the following image:



The last and final task is to update the chart when the client receives a ping back from the server with new set of stock values. Now, the parameters of the chart would change as there will be some change in the domain of the values of either axes. So we need to redraw the axes with new set of values and update the chart with new data. Following function performs all of these tasks:

```
function redrawLineChart() {
  setChartParameters();
  svgElement.selectAll("g.y.axis").call(yAxisGen);
  svgElement.selectAll("g.x.axis").call(xAxisGen);
  svgElement.selectAll("." + pathClass)
    .attr({
      d: lineFun(lineData)
    });
}
```

Now you should be able to see the chart updating after every 10 seconds. The chart will also be updated when a new company is selected from the companies select control.

CONCLUSION

In this article, we saw how to leverage the rich features supported by modern browsers to build a basic real-time graphics chart application using D3 and SignalR. As we go on exploring, there are vast number of possibilities if we use these technologies creatively. Such features make the sites interactive and also help in catching attention of the users very easily. I encourage you to checkout more features of the technologies we discussed in this article, as they have numerous capabilities to build feature rich and sophisticated applications ■

Download the entire source code from our GitHub Repository at bit.ly/dncm14-signalr-d3

AUTHOR PROFILE

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