ISSUE 07 | JULY-AUG 2013 EDITION

DNCMagazine

www.dotnetcurry.com

ANNIVERSARY

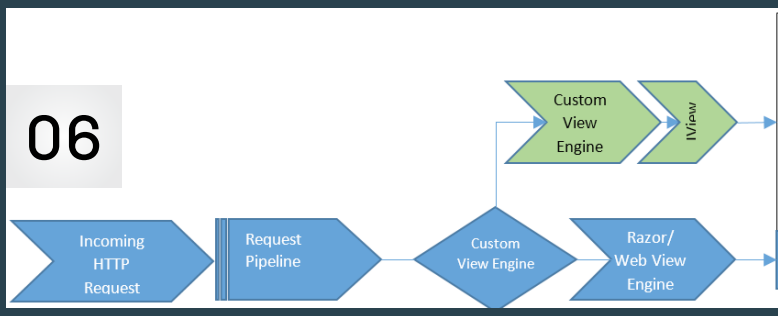
GIVING AWAY .NET TRAINING, BOOKS AND SOFTWARE

58

16

GLIMPSE

Explore the Internal working of your ASP.NET Web Application using Glimpse



06

ANDROID CLIENT FOR TFS

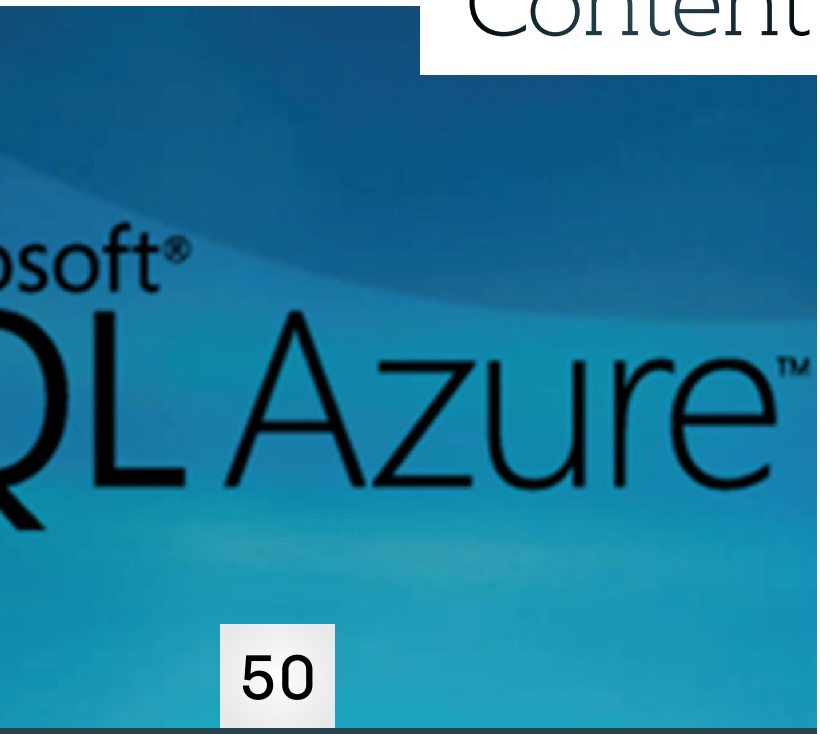
An innovative use case for the newly launched TFS OData Services

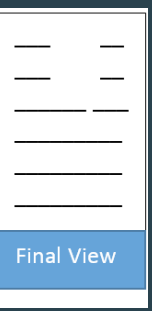
SORTABLE, PAGED DATATABLE

Building a sortable, paged datatable using Knockout JS & ASP.NET Web API

62

Content



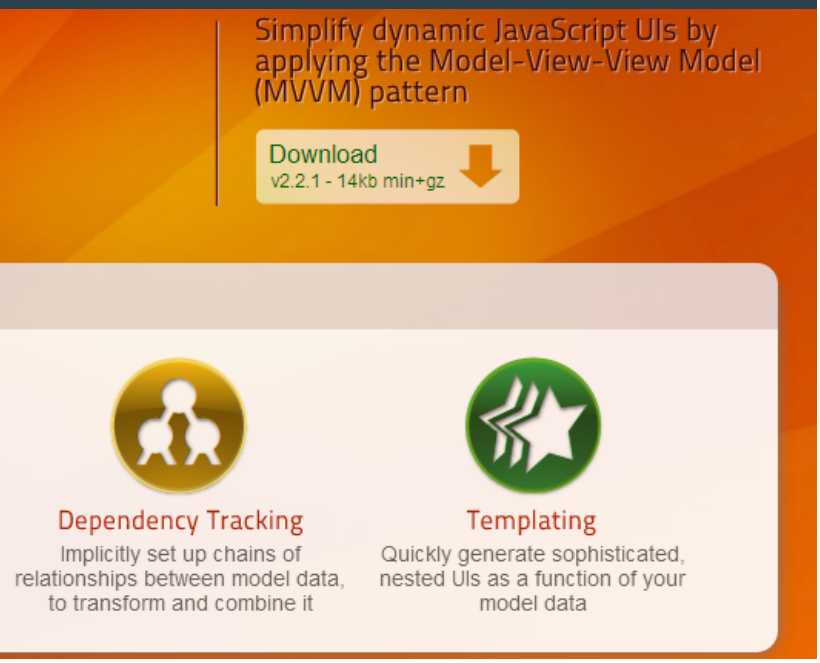


AZURE SQL REPORTING

Migrating On-Premises Database to Cloud and using SQL Reporting

CUSTOM VIEW ENGINE

A custom ASP.NET MVC view engine that loads views defined in a SQL database



www.dotnetcurry.com

dnc mag

Editor-In-Chief • Sumit Maitra
sumitmaitra@a2zknowledgevisuals.com

Editorial Director • Suprotim Agarwal
suprotimagarwal@dotnetcurry.com

Art & Creative Director • Minal Agarwal
minalagarwal@a2zknowledgevisuals.com

Contributing Writers • Filip W, Mahesh Sabnis, Pravinkumar Dabade, Raj Aththanayake , Subodh Sohoni, Sumit Maitra, Suprotim Agarwal

Advertising Director • Satish Kumar
business@dotnetcurry.com

Writing Opportunities • Carol Nadarwalla
writeforus@dotnetcurry.com

NEXT ISSUE - 1st September, 2013

POWERED BY

a2z Knowledge Visuals

A Big THANK YOU to our wonderful authors who have selflessly contributed some awesome .NET content past one year. Here are our champions: Subodh Sohoni, Raj Aththanayake, Filip Ekberg, Filip W, Gregor Suttie, Gouri Sohoni, Govind Kanshi, Mahesh Sabnis, Mehfuz Hossain, Omar Al-Zabir, Pravinkumar Dabade, Anoop Madhusudanan, Fanie Reynders, Jonathan Channon, Niraj Bhatt, Piotr Walat and Shiju Verghese

We strive to deliver the best possible information about a given technology and present it in the most comprehensive and usable way. We'll continue doing so on a regular basis. However, to do this, we need your support and your input.

Reach out to us on twitter with our hashtag #dncmag or email me at suprotimagarwal@dotnetcurry.com with your ideas and feedback.

A Note From the Founder

Without YOU, this magazine would not exist! On behalf of the entire DNC .NET Magazine team, **Thank You** for reading us.

The DNC .NET Magazine is one year old! Some of you are regular visitors of our .NET websites www.dotnetcurry.com and www.devcurry.com that has been online past 6 years, & helping millions of .NET devs with some awesome tutorials. But we always felt something was missing. About fifteen months ago, the two of us (Sumit and I) sat together wondering how to present high quality programming related content in an *offline format*, specifically targeted towards Enterprise Developers and Architects. This would give our readers the added flexibility to access some of our exclusive .NET content in the form of a PDF or may be, even on a tablet.

The Idea behind this magazine was to include selected, conceptually deep, programming topics covering all aspects of the software development lifecycle. Since both of us were from Microsoft background, MS Technologies was our first go to. The idea of a Digital magazine was thus born!

Beyond turning one, the DNC .NET Magazine has other things to celebrate. At the time of writing, we have published 56 Exclusive articles, 6 interviews with industry experts, over 40000 subscribers, downloaded over 100,000 times, tweeted over 1500 times and not to mention, the countless accolades and some constructive criticisms we have received from you.

Thank you! Now it's time for you to get something back. To celebrate one year of Your magazine, we are doing a Giveaway of .NET Training, Books and Software. Make sure you participate to win some cool goodies. Good luck!

Be prepared to learn! This year we're going to be better than the previous one as we'll strive to involve more authors with different perspectives, as well as interesting peeks into the thoughts and beliefs or our beloved industry stalwarts.

Celebrate with us!

Suprotim Agarwal &
Sumit Maitra

INFRAGISTICS
DESIGN / DEVELOP / EXPERIENCE

pluralsight
hardcore developer training

O'REILLY

To participate in the Giveaway

CLICK HERE

BUILDING A DATA DRIVEN VIEW ENGINE IN ASP.NET MVC

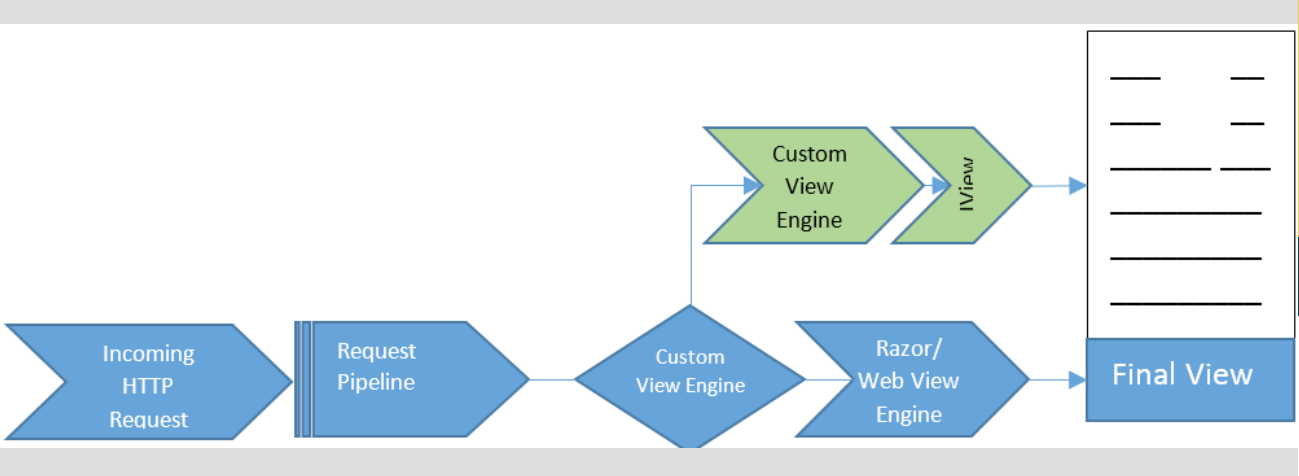
"Sumit Maitra explores a major extensibility point in ASP.NET MVC, the View Engine. In this article he builds a working view engine that loads views defined in a SQL database"

The ASP.NET MVC framework, as we know, is highly pluggable, allowing you to easily swap out pieces of the framework with custom implementation. One such piece in the framework is the View Engine. Out of the box, ASP.NET MVC comes with two view engines: the Razor View Engine and the Web Forms View Engine. Usually these default engines are enough to fulfill most of your view requirements. Both of them support decent server side template options for dynamically generating fields. However there are some scenarios that you may end up having, where the flexibility of server side template is not enough. Scenarios could be like CMS systems that actually allow you to create UI or systems that generate dynamic data entry forms where the form definition is defined by the system itself.

Of course you could build a view engine just because you don't like the verbosity of either of the two default engines! Today we will see how we can build a web app that needs to save data form definitions in a database and then generate the views at runtime for data entry. But before we do that, let's see what constitutes a view engine in ASP.NET MVC and how we can go about building one.

THEORY BEHIND A CUSTOM VIEW ENGINE

A Custom View Engine in ASP.NET MVC plugs into the MVC pipeline towards the end of the request cycle. If you refer to Steve Sanderson's excellent MVC Pipeline Poster, you will notice that once the Controller processes the Action, depending on whether the Return Value is ViewResult or not,



the View Engine is invoked.

The View Engine returns a View instance that implements the IView interface.

We could have a Custom View Engine that returns a Razor view, but if you are doing that, you could probably do with HTML Helpers and don't need a Custom View Engine. So if you are implementing a View Engine, you ideally implement an IView as well. The final picture of this last leg of a Request's journey is as you see above in the graphic.

As you can see, the Incoming HTTP Request passes through various stages in the Request Pipeline before finally hitting the check for ViewEngine. If a Custom View engine is available, it runs through the view engine and if the custom view engine can render the view, it renders the appropriate View. If not, it passes the request to the next view engine available. With the picture of the Custom View Engine vis-à-vis the Request pipeline clear, let's go ahead and build a view engine and a custom IView implementation.

GETTING STARTED WITH A CUSTOM VIEW ENGINE.

To get started with our View Engine, we'll start off with a standard ASP.NET MVC 4 Project using the Internet Template. I am starting off with .NET Framework 4.5. You can select either 4.0 or 4.5. Only the usage of AntiXss Encoder will vary as we will see later.

Implementing the IViewEngine interface.

Step 1: To start off with the View Engine, we'll add a new folder, I call it 'Core' feel free to name it as you deem fit.

Step 2: Next we add a class called DbDrivenViewEngine and implement the IViewEngine interface in it.

The IViewEngine has the following methods that need to be implemented:

```
public ViewEngineResult
FindPartialView(ControllerContext
controllerContext, string
partialViewName, bool useCache){
  …
}
public ViewEngineResult
```

```
FindView(ControllerContext controllerContext, string
viewName, string masterName, bool useCache)
{
 …
}

public void ReleaseView(ControllerContext controllerCon-
text, IView view)
{
 …
}
```

The FindPartialView method is called when the Controller is looking to return a Partial View with the given Name.

The FindView method is called when Controller is looking for a View with a given Name.

ReleaseView method is provided to release any resources that the ViewEngine might be holding on to.

Step 3: Given the ControllerContext, the viewName, the masterName and the useCache parameter, the FindView (or FindPartialView) method has to determine if it should go forth and try to render the view or pass the opportunity to other view engines in queue!

If it chooses to render the view, it will create an instance of ViewEngineResult and pass it two parameters, the IView implementation and the IViewEngine implementation.

If it chooses to pass on the opportunity to the remaining view engines, it will create an Instance of the ViewEngineResult with an array of strings, specifying the locations where it failed to find the required view.

The barebones implementation for this is as follows:

```
public ViewEngineResult FindView(
 ControllerContext controllerContext,
 string viewName,
 string masterName,
 bool useCache)
{
 if (viewName.Contains("Dynamic"))
 {
  return new ViewEngineResult(new DbDrivenView(),
this);
 }
 else
 {
  return new ViewEngineResult(
   new string[]
    { "No Suitable view found for 'DbDataViewEngine',
please ensure View Name contains
       'Dynamic'" });
 }
}
```

Here we see there is a minimal check to see if the viewName contains the word 'Dynamic' and if it does, it tries to render it by using a new instance of DbDrivenView(). This is a class we will create in the next step, so take it for its face value at the moment.

If the View Name does not contain the word 'Dynamic', we send back an array of string with one message as of now indicating that we didn't find a view and the probable solution. This message is used ONLY when MVC falls through all the view engines and is unable to render using any one of them.

Now let's implement the IView.

Implementing the IView interface

The DbDrivenView class that we saw passed into the ViewEngine Result, is an implementation of the IView interface. The IView interface just has the Render method. The simplest implementation of the Render method is shown below.

```
public class DbDrivenView : IView
{
 public void Render(ViewContext viewContext, TextWriter
writer)
 {
  writer.Write("Hello World");
 }
}
```

All it does is write 'Hello World' to the TextWriter. We save this also under the Core folder.

Running the Rudimentary Custom View Engine

Now that we've got the ViewEngine and the View in place, we can technically 'use' it in an MVC 4 application.

Step 1: Registering the ViewEngine. To register our ViewEngine, we use the following code in our Application_Start event in the Global.asax file.

```
ViewEngines.Engines.Insert(0, new DbDrivenViewEngine());
```

As we can see, we are inserting it on the top of the already existing array of ViewEngines. This way, MVC will call our View Engine First.

Step 2: Adding an Action called 'Dynamic'

Next we add an Action called Dynamic that returns a ViewResult. So in the HomeController, we add the following code.

```
public ViewResult Dynamic()
{
    return View();
}
```

Note that the View() method is called without Parameters, this by default implies that the View Name is the same as the Action Method Name (= Dynamic). So call to this Action method will trigger our ViewEngine.

Step 3: Now if we run the application, we get our Default ASP.NET Home page. This shows that our default View Engine is in place and working fine. To trigger our ViewEngine, we navigate to the /Home/Dynamic URL and you should see 'Hello World'.

Nice!!! There you go, you have a custom view engine! Go home now, we are done here!!!

Okay, I was kidding, we are far from done. Though most text books in 'advanced ASP.NET series', would have left you here, we have just gotten started. Tighten your seatbelts, its going be one heck of a ride!

SAVING CUSTOM VIEW DEFINITION IN DATABASE

Let us now decide how we would like to save our Dynamic View. This is where it can get as complex as you want it to be. For sake of the Demo, we'll keep it simple.

One naming convention I am following here is, anything related to building the View and the ViewEngine, I am calling it a part of 'Core'. So the View and ViewEngine itself were a part of core. So are the Entities that help define the core Views, so the following entities, the controllers and views required to generate those entities, will all be referred to as part of the Core.

The Entities

- We will have a DataForm Class that will store
  - o Name, Description, Template and list of DataFields.
  - o The Template string will store an HTML string into which the dynamic view will be injected. Essentially the Template can be used to save the 'Master Page' of our View. We will later see what keywords we can use to serve as the placeholder for the dynamically generated HTML.

```
public class DataForm
{
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
    [Required]
    public string Description { get; set; }

    public string Template { get; set; }

    public List<DataField> Fields { get; set; }
}
```

- Each DataField will save
  - o The Field Type like Input Text Boxes, Selection Options, Check Boxes, Images etc.
  - o It will store the display label for these and the Max Length
  - o Flag indicating whether these fields are mandatory or not.
  - o We also have a 'Foreign Key' reference to a Data

```
public class DataField
{
    public int Id { get; set; }
    [Required]
    public string FieldName { get; set; }
    [Required]
    public string DisplayLabel { get; set; }
    [Required]
    public FieldType DisplayType { get; set; }
    [Required]
    public bool IsMandatory { get; set; }

    public int FormId { get; set; }
    [ForeignKey("FormId")]
    [ScriptIgnore]
    public DataForm ParentForm { get; set; }
}
```

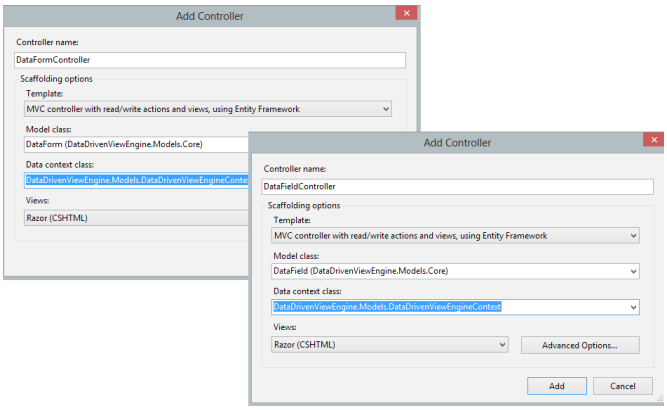
- The FieldType enum is defined as follows. We can save the

class file in the Models\Core folder itself

```
public enum FieldType
{
    TextBox,
    TextArea,
    DropDown,
    CheckBox,
    Label
}
```

Saving a View – The Core Controllers and Views

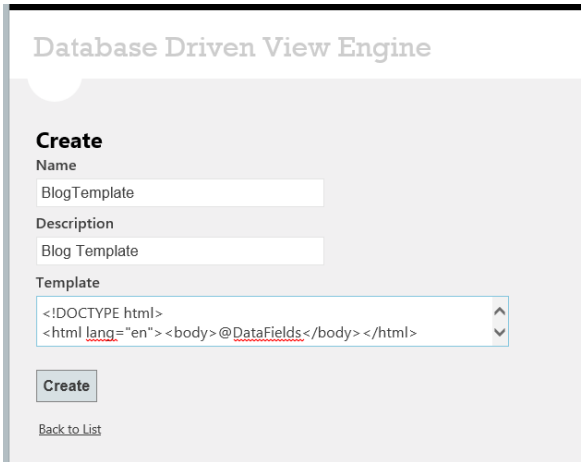
Once the Entities are setup, we'll create a Core folder and add the DataForm and DataField Controllers using the MVC Tooling. We'll use Entity Framework as the backend.



That completes the CRUD support for the Core Form Entities. So we can now go and define some Forms for ourselves.

Creating a DataForm

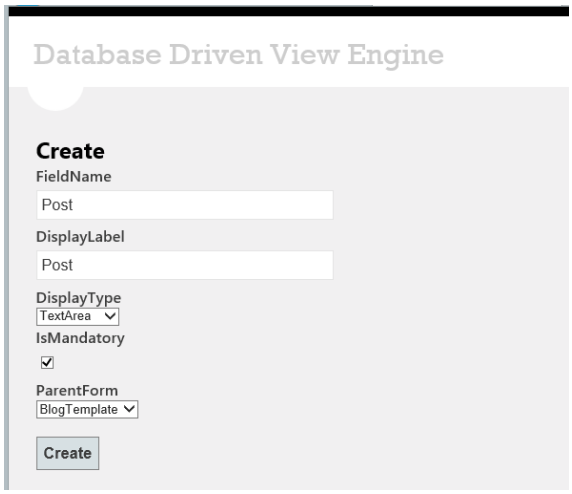
Our First Template will be a Blog Template that we define as follows



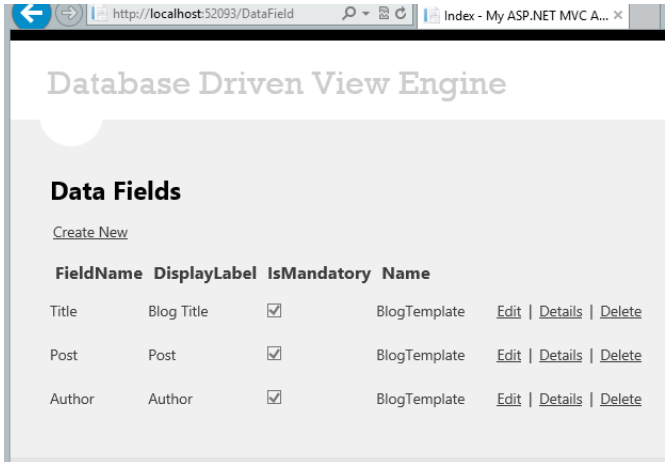
As we can see above, the Template field is most interesting. We've specified a rudimentary HTML skeleton with the key word '@DataFields'. This key word will be replaced by the markup for the Fields that are generated at runtime. So next let us see how we define the Fields themselves.

Creating Data Fields for the Form

For the Blog Template we will have three fields - Title, Author and Post. Following shows creation of the "Post" field



Once all three fields are created, the DataField Index page is as follows:



So now we have defined one form in our database. It is for creating Blog Post entries and had three entry fields: Title, Post and Author. This is an oversimplification for the purpose of this article, but as I mentioned, we can make this as complicated as we want. Let's continue with the proof of concept.

LOADING VIEW DEFINITION INTO VIEW ENGINE

Now that our View has been defined and stored in the Database, let's see how we can retrieve it and render it into a View.

Updating the Custom View

1. In the DbDrivenView class, declare an instance of DataDrivenViewEngineContext.

2. Next add a Constructor that takes in the View Name. This will have to match our Template's Name property.

3. In the Render method, we'll use the View Name property to pull up the Template and its Fields. Next we will loop through the Fields and render them using the TextWriter that was passed on to the Render method.

The updated Class is as follows

```
public class DbDrivenView : IView
{
 DataDrivenViewEngineContext dbContext = new
              DataDrivenViewEngineContext();
 string _viewName;
 public DbDrivenView(string viewName)
 {
  if (string.IsNullOrEmpty(viewName))
  {
   throw new ArgumentNullException("viewName", new
     ArgumentException("View Name cannot be null"));
  }
  _viewName = viewName;
 }
 public void Render(ViewContext viewContext, TextWriter
   writer)
 {
  DataForm form = dbContext.DataForms.Include("Fields").
   First(f => f.Name == _viewName);
  HtmlTextWriter htmlWriter = new
           HtmlTextWriter(writer);
  htmlWriter.RenderBeginTag(HtmlTextWriterTag.Div);
  foreach (var item in form.Fields)
  {
   htmlWriter.RenderBeginTag(HtmlTextWriterTag.Div);
   htmlWriter.WriteEncodedText(item.DisplayLabel);
   htmlWriter.RenderBeginTag(GetHtmlRenderKey(item.
    DisplayType));
   htmlWriter.RenderEndTag();
   htmlWriter.RenderBeginTag(HtmlTextWriterTag.Div);
  }
  htmlWriter.RenderEndTag();
 }
 private HtmlTextWriterTag GetHtmlRenderKey(FieldType
 fieldType)
 {
  switch (fieldType)
  {
   case FieldType.TextBox:
    return HtmlTextWriterTag.Input;
   case FieldType.TextArea:
    return HtmlTextWriterTag.Textarea;
   case FieldType.DropDown:
    return HtmlTextWriterTag.Select;
   case FieldType.CheckBox:
    return HtmlTextWriterTag.Input;
   case FieldType.Label:
    return HtmlTextWriterTag.Caption;
   default:
    return HtmlTextWriterTag.Unknown;
  }
 }
}
```

Updating the Custom View Engine

Now that we have a constructor that takes one parameter, we will have to update the View Engine to pass the parameter. We will also change the Condition to use our view. Instead of checking for the ViewName to be Dynamic, we'll check if the Route has the string "Dynamic".

```
public ViewEngineResult FindView(
    ControllerContext controllerContext,
    string viewName, string masterName, bool useCache)
{
 if (controllerContext.RouteData.Values.
   ContainsValue("Dynamic"))
 {
     return new ViewEngineResult(new
DbDrivenView(viewName),
     this);
 }
 else
 {
  return new ViewEngineResult(new string[]
  { @"No Suitable view found for 'DbDataViewEngine',
   please ensure View Name contains 'Dynamic'"
  });
 }
}
```

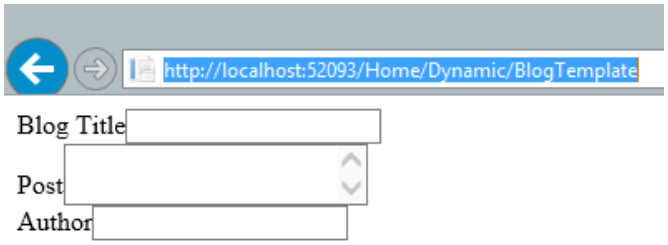
Once these two changes are in place, we can run the application to check it out. The only condition is that the Route should have the value Dynamic in it. So we add an Action

method in HomeController called Dynamic with the input being the view name that we want to render

```
public ViewResult Dynamic(string id)
{
    return View(id);
}
```

Now run the application and navigate to the URL http://localhost:[your port]/Home/Dynamic/BlogTemplate/

Voila! We get this!



If you just cringed and went "eww..." at the HTML nightmare above, I hear you. Also if you were paying attention you'll realize we didn't use the DataForm's Template field value. Let's see how we can stuff our rendered HTML into the Template and then how to make the template better.

Adding some Styling and Eye Candy

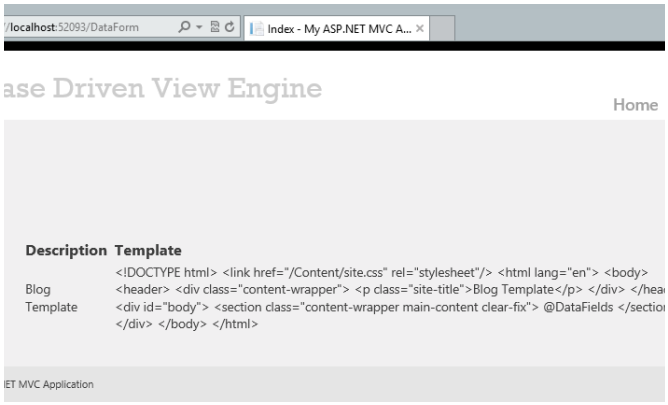
First we update the Custom View's Render method to use a StringWriter so that we can write the dynamic string independent of the TextWriter instance sent to us. Once we are done writing to the StringWriter, we replace the "@DataFields" token with this generated string. Finally we write this string out to the TextWriter instance of the Render method.

```
public void Render(ViewContext viewContext, TextWriter
writer)
{
 DataForm form = dbContext.DataForms.Include("Fields").
 First(f => f.Name == _viewName);
 var sb = new StringBuilder();
 var sw = new StringWriter(sb);
 using (HtmlTextWriter htmlWriter = new
 HtmlTextWriter(sw))
 {
  htmlWriter.RenderBeginTag(HtmlTextWriterTag.Div);
  foreach (var item in form.Fields)
  {
   htmlWriter.RenderBeginTag(HtmlTextWriterTag.Div);
   htmlWriter.WriteEncodedText(item.DisplayLabel);
   htmlWriter.RenderBeginTag(GetHtmlRenderKey(item.
   DisplayType));
   htmlWriter.RenderEndTag();
   htmlWriter.RenderBeginTag(HtmlTextWriterTag.Div);
  }
  htmlWriter.RenderEndTag();
 }
 writer.Write(form.Template.Replace("@DataFields",
sb.ToString()));
}
```

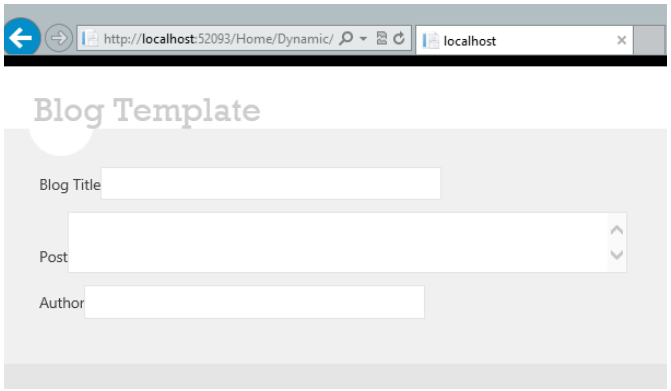
Now we go back to DataForm template definition and update the Template to be as follows

```
<!DOCTYPE html>
<link href="/Content/site.css" rel="stylesheet"/>
<html lang="en">
 <body>
  <header>
   <div class="content-wrapper">
    <p class="site-title">Blog Template</p>
   </div>
  </header>
  <div id="body">
   <section class="content-wrapper main-content
    clear-fix">
   @DataFields
   </section>
  </div>
 </body>
</html>
```

As we can see, we have utilized the existing CSS that we get in the ASP.NET Project template.



And the Result? Here you go! Nice eh!



SUBMITTING THE FORM DATA

Well, we have the Form, we have some styling, but we don't have a way to submit the data that user fills in the form. We could do multiple things here:

a. We could update the Form Fields to add a button and then do AJAX Posts or
b. Add another property to our Form called Submit Data that will take a URL to which we should Submit Data. Let's see what it takes to submit the Form Data.

Updating the DataForm Entity

Let's update the DataForm entity and also visit some of the things I didn't talk about earlier, but I already did to get the sample working.

The modified DataForm Entity

I have highlighted the changes in the code below:

1. We have a backing field called _template that stores the Template html string.

2. The Template property is decorated with the [AllowHtml] attribute. If we don't do this then MVC will throw an error saying suspicious characters detected. This is to prevent XSS hacks via unprotected Inputs. In our case, we genuinely need HTML to come through, so we'll let it come in, but before assigning it to the backing store, we will sanitize it using HttpUtility.HtmlEncode(...) method.

3. Since the data being saved is HtmlEncoded, when returning the data, we've to return it HtmlDecoded.

```
public class DataForm {
    string _template;
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
    [Required]
    public string Description { get; set; }
    [DataType(DataType.MultilineText)]
    [AllowHtml]
    public string Template     {
        get
        {
            return HttpUtility.HtmlDecode( _template);
        }

        set
        {
            string sanitized = HttpUtility.
            HtmlEncode(value);
            _template = sanitized;
        }
    }
    public List<DataField> Fields { get; set; }
    public string SubmitUrl { get; set; }
    public string SubmitName { get; set; }
}
```

4. To make our Encoding more secure, we will use the AntiXSS Library to do the Encoding. For that, we don't need any change to the HttpUtility.HtmlEncode(...) call. Instead we plug in the AntiXSS library in the Web.config as follows

```
<httpRuntime
 targetFramework="4.5"
 encoderType="
 System.Web.Security.AntiXss.AntiXssEncoder,
 System.Web, Version=4.0.0.0, Culture=neutral,
              PublicKeyToken=b03f5f7f11d50a3a"/>
```

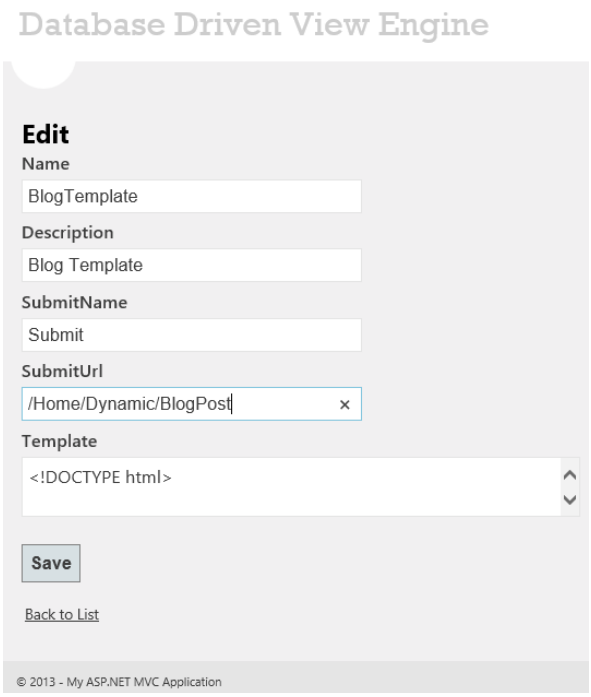
In .Net 4.5 you will have the httpRuntime node with the targetFramework attribute already. Add the encoderType as shown above to plug in the AntiXssEncoder, as the default Encoder. HttpUtility will now use this.

Point 3 and 4 here are actually required to do the demo so far. So I had already done this to run our Demo.

5. Now we add two new properties SubmitUrl and SubmitName. They are self-explanatory, one stores the Label of the Button and the other Stores the URL to which we shall post the Data.

With these changes clarified, let's go to our Custom View generator and see what it takes to render the additional markup to make our View into a Submit-able HTML Form.

6. We update the respective Index/Create and Edit pages to Read/Create and Update the new fields. The following screen shows the edit screen for our current Template.



Note: When you add new fields to a Database that you are posting data to using Entity Framework, EF will protest when the Schema changes. You have to use EF Migrations to update your database after you add properties to the DataForm entity.

Updating the Custom View – DbDrivenView

With the new fields in place, we use them to render the appropriate Form tags and the Submit button to post the data to the controller.

I have tagged portions of the code below for reference.

#1: We add the Form Tag and add the action and method attributes to it. The action attribute gets the value of the 'SubmitUrl' property. We have a check for Empty on the SubmitUrl property so if there is no SubmitUrl, then we don't create the Form. Section **#3** does the same check before adding the Form's end tag.

#2: The fields that we are showing in our view must have an 'id' and 'name' set. We were not utilizing the FieldName property of our DataField objects. Now we will use this to populate the Id and Name attribute of the Field.

Notice a peculiarity of HtmlWriter's AddAttribute property. Since the writer is forward only cursor you have to declare the attributes before you render the begin tag using RenderBeginTag method.

#4: If the SubmitName property is populated we add the Submit button to the Form as well.

```
public void Render(ViewContext viewContext, TextWriter
writer) {
 DataForm dataForm = dbContext.DataForms.
Include("Fields").First(f => f.Name == _viewName);
 var sb = new StringBuilder();
 var sw = new StringWriter(sb);
 using (HtmlTextWriter htmlWriter = new
HtmlTextWriter(sw))  {
  // #1 Begins
  if (!string.IsNullOrEmpty(dataForm.SubmitUrl))
  {
   htmlWriter.AddAttribute("action",
   dataForm.SubmitUrl);
   htmlWriter.AddAttribute("method", "post");
   htmlWriter.RenderBeginTag(HtmlTextWriterTag.Form);
  }
  // #1 Ends
  htmlWriter.RenderBeginTag(HtmlTextWriterTag.Div);
  foreach (var item in dataForm.Fields)   {
   htmlWriter.RenderBeginTag(HtmlTextWriterTag.Div);
   htmlWriter.WriteEncodedText(item.DisplayLabel);
   // #2 Begins
   htmlWriter.AddAttribute(HtmlTextWriterAttribute.Id,
   item.FieldName);
   htmlWriter.AddAttribute(HtmlTextWriterAttribute.Name,
   item.FieldName);
   // #2 Ends
   htmlWriter.RenderBeginTag(GetHtmlRenderKey(item.
   DisplayType));
   htmlWriter.RenderEndTag();
   htmlWriter.RenderBeginTag(HtmlTextWriterTag.Div);
  }
  // #3 Begins
  if (!string.IsNullOrEmpty(dataForm.SubmitUrl))   {
   htmlWriter.RenderEndTag();
  }
  // #3 Ends
  // #4 Begins
  if(!string.IsNullOrEmpty(dataForm.SubmitName))   {
   htmlWriter.AddAttribute("type", "submit");
```

```
    htmlWriter.AddAttribute("value",
    dataForm.SubmitName);
    htmlWriter.RenderBeginTag(HtmlTextWriterTag.Input);
    htmlWriter.RenderEndTag();
   }
   // #4 Ends
   htmlWriter.RenderEndTag();
  }
  writer.Write(dataForm.Template.Replace("@DataFields",
sb.ToString()));
}
```

Updating the Controller to receive the HttpPost

We make two changes to the HomeController class. First, we decorate the existing Dynamic action method with the attribute [HttpGet] as follows

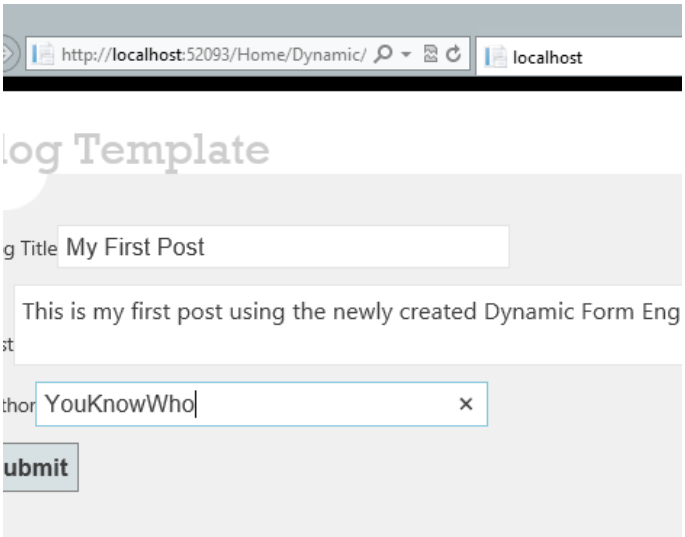
```
[HttpGet]
public ViewResult Dynamic(string id)
{     return View(id); }
```

Next we add another Dynamic action method for handling the HttpPost as follows

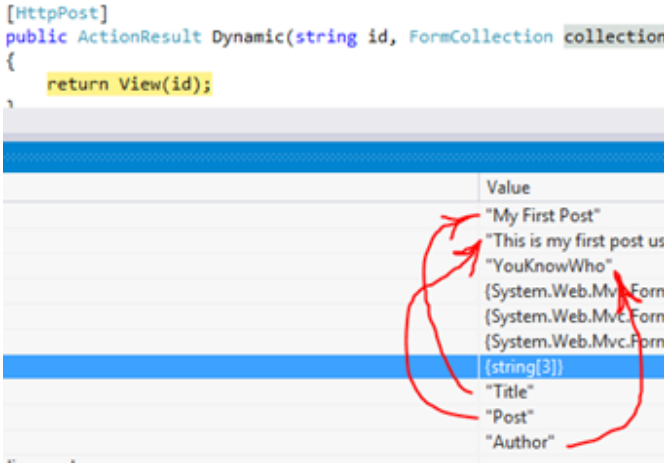
```
[HttpPost]
public ActionResult Dynamic(string id, FormCollection
collection) {
     return View(id);
}
```

Testing it Out

1. To test the Posting of data, we put a breakpoint in the newly added Action method and navigate to our View.



2. Populate some data and hit Submit. As we hit the breakpoint, we add the FormCollection to the watch list and expand it. We can see the collection has three keys corresponding to the three fields with the exact same names as provided in FieldName of our Dynamic fields. Lastly we see that the values that we posted from above, have come through correctly in the controller! Awesome!



IN CONCLUSION – LOTS MORE TO DO

We saw how to build a new Custom View Engine, Custom IView implementation, we built a dynamic form and we used the Dynamic Form to submit data back as well. However we have only scraped the surface of building a full-fledged View Engine. We haven't touched upon vital pieces like Caching, DataBinding and Validation. We leave these for another day or if you are getting into the groove, you can send pull requests to our GitHub repo and send across your additions for publication.

We conclude this chapter of View Engine Creation here today. Even if you don't need to do this ever, hope it gave you a clearer picture of how the View Engines in the MVC Framework work. ■

Download the entire source code from our GitHub repository at bit.ly/dncm7-ddve

Sumit is a .NET consultant and has been working on Microsoft Technologies for the past 12 years. He edits, he codes and he manages content when at work. C# is his first love, but he is often seen flirting with Java and Objective C. You can Follow him on twitter @sumitkm and read his articles at bit.ly/KZ8Zxb

A GLIMPSE INTO THE GUT OF YOUR ASP.NET APP

ASP.NET Architecture MVP Suprotim Agarwal explores the Glimpse package that helps you look at the internal working of your ASP.NET Web Application, in the browser

Some of us may have heard or used MiniProfiler, a profiling library for your ASP.NET applications, from the StackExchange team. We carried an article on dotnetcurry.com on it. Today we are going to review another OSS project called Glimpse from a team led by Anthony vander Hoorn and Nik Molnar. Glimpse recently launched v1.4 with a new Heads up Display (HUD) for the profile information!

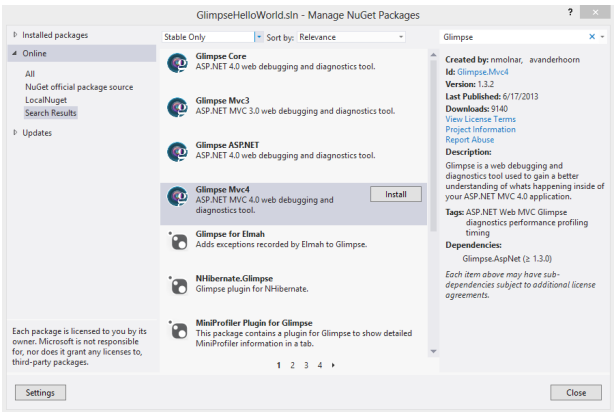
On the outset, Glimpse seems to have the same or overlapping functionality as MiniProfiler, but after playing with for about 5 minutes, I realized, Glimpse provides significantly more insight into the happening of your Application than MiniProfiler. So without further ado, let's dive in and see what Glimpse can do for us.

GETTING STARTED WITH GLIMPSE

To start off with I wanted a project with some functionality already. I thought it would be good to use the same one that we used for our MiniProfiler. So I went ahead and downloaded the Source code as Zip from the Github Project.

Glimpse is distributed via Nuget packages, so the easiest way to get started with it is to fire up the Manage Nuget Packages dialog and search for 'Glimpse'.

> ..after playing with Glimpse for about 5 minutes, I realized, it provides significantly more insight into the happening of your Application than MiniProfiler



Depending on the type of project you want to add Glimpse to, you can select Mvc3, Mvc4 or in case of WebForms, Glimpse ASP.NET. In our case, I selected Glimpse Mvc4, this installs the Glimpse ASP.NET and Glimpse Core packages. The following packages get installed

```
<package id="Glimpse" version="1.4.2"
targetFramework="net40" />
<package id="Glimpse.Ado" version="1.4.0"
targetFramework="net40" />
<package id="Glimpse.AspNet" version="1.3.0"
targetFramework="net40" />
```

Changes under the hood

The Glimpse architecture is based on HttpHandlers and HttpModules. So Glimpse installation basically implies registering the required modules. Let's look at the web.config changes:

New Glimpse section: We have a new Glimpse section for glimpse specific settings.

```
<configSections>
  …
  <section name="glimpse" type="Glimpse.Core.
  Configuration.Section, Glimpse.Core" />
</configSections>
```

The Handler and Module Registration: Next we have the Glimpse module and the Glimpse.axd registered.

```
…
  <httpModules>
    <add name="Glimpse" type="Glimpse.AspNet.HttpModule,
    Glimpse.AspNet" />
  </httpModules>
  <httpHandlers>
    <add path="glimpse.axd" verb="GET" type="
    Glimpse.AspNet.HttpHandler, Glimpse.AspNet" />
  </httpHandlers>
</system.web>
…
<modules runAllManagedModulesForAllRequests="true">
  <add name="Glimpse" type="Glimpse.AspNet.HttpModule,
Glimpse.AspNet" preCondition="integratedMode" />
</modules>
…
 <add name="Glimpse" path="glimpse.axd" verb="GET"
type="Glimpse.AspNet.HttpHandler, Glimpse.AspNet"
preCondition="integratedMode" />
</handlers>
```

The Glimpse Policy section: We will look at policies in a bit. This section is for configuring Glimpse specific settings, that include when should Glimpse record data and when not, as well as when should the data be displayed, so on and so forth.

```
<glimpse defaultRuntimePolicy="On"
endpointBaseUri="~/Glimpse.axd">
 <!-- If you are having issues with Glimpse, please
include this. It will help us figure out whats going
on. <logging level="Trace" />-->
 <!-- Want to use Glimpse on a remote server? Ignore
the LocalPolicy by removing this comment.
 <runtimePolicies>
  <ignoredTypes>
   <add type="Glimpse.AspNet.Policy.LocalPolicy,
Glimpse.AspNet"/>
  </ignoredTypes>
 </runtimePolicies>-->
</glimpse>
```

From the configuration point of view, Glimpse is all set, unlike

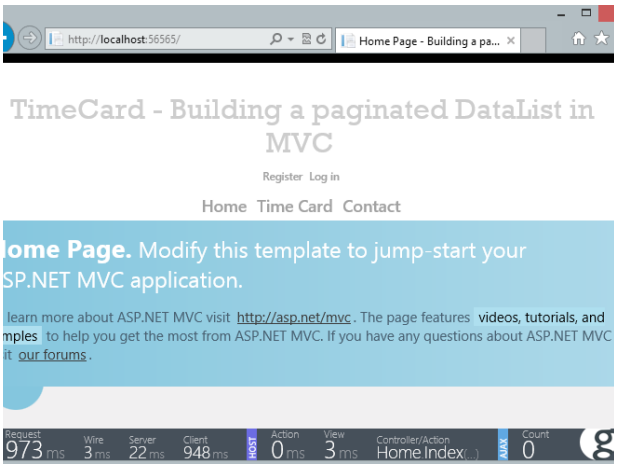
MiniProfiler, we don't have to get it going with a call in the Global.asax. So far zero code required.

Starting Glimpse

We can run the application now and navigate to /Glimpse.axd. This will give us the following page

Starting Glimpse is as simple as hitting the 'Turn Glimpse On' button.

Now when we navigate back to the home page, we should see the Glimpse HUD at the bottom of the page. If it's not expanded, click on the 'g' logo to expand it.

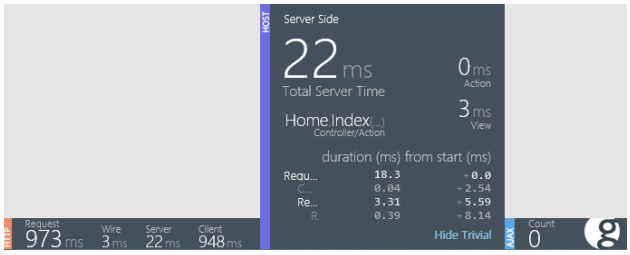


THE HUD

As we can see above, the Heads Up Display panel from Glimpse gives us a quick view of what's going on with our Request (Glimpse as it's subtly obvious, work per-HTTP Request). In the panel we see three sections:

1. HTTP: Shows the complete request took 973 ms to complete, of which 3ms was 'on the wire', meaning the time between the client requesting and server receiving the request. Next 22 milliseconds were spent by the server to process the request and finally 948 seconds by the client to receive and render the Server's response.

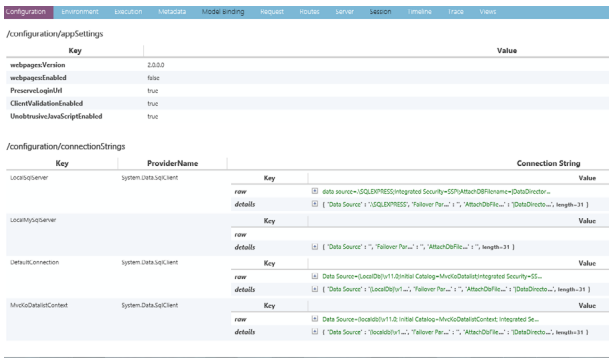
2. Host: Shows details of activities on the server. We can see here that it took almost no time to resolve the correct Action and the View rendering took 3 ms. If we hover over it, we'll see a little more details



3. AJAX: The final section shows the number of AJAX requests. This page had 0 requests but when we do have AJAX requests, hovering over the AJAX section will show us more details for each AJAX request made.

So that was the HUD for Glimpse and we have gotten off to a good start. Still it looks similar to MiniProfiler. Well if MiniProfiler blew you away, Glimpse is going to put you in Orbit!

THE GLIMPSE PANEL



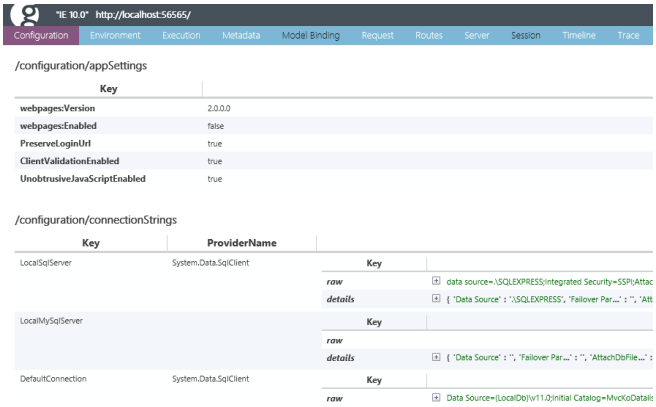
From the HUD display if we click on the Glimpse logo at the bottom right corner, the Glimpse panel is launched! This is not a browser plugin rather a dynamically injected CSS/JS panel full or metrics collected about your ASP.NET Application. Blown away yet?

Well, at first glance the data shown looks similar to what

the Firebug plugin shows us, but there is a lot more than Firebug. Fact is, Glimpse by the virtue of being on the Server and the Client digs out way more data than Firebug and except for Request information, there is hardly any overlap. Of the default Tabs above, we will look at Configuration, Environment, Routes, Timeline, Trace and Views in details today.

With MVCs convention-over-configuration approach a lot of times, things work 'magically' and we are not sure about the exact mechanics. Glimpse serves as a valuable tool to see what is happening behind the scenes by unravelling how the conventions are interpreted.

Configuration



The Configuration 'tab' pulls data out of our application's Web.config as well as Machine.config. Essentially every configuration setting applicable to the app is pulled in. For example, above I see that once upon a time I had installed MySqlServer on my machine and it seems to have registered a provider on my system.

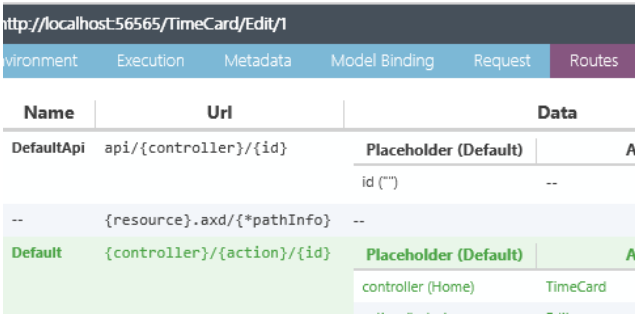
Among other things that it grabs, you can see the Custom Error settings, HTTP Modules and Handlers configured.

Environment

The environment gives us details of the Machine, Web Server, .NET Framework, Process ID as well as Assemblies in use among other things. For the assemblies loaded it shows, version, culture, whether it's from GAC & if they're Full Trust.

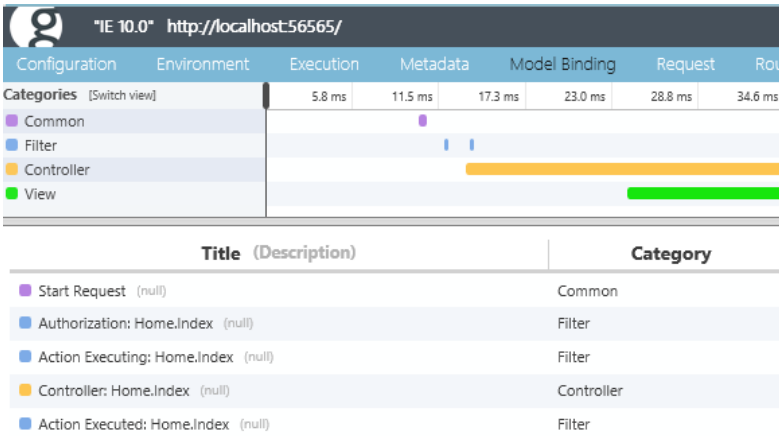
Routes

The Routes tab shows us how the routing framework picked the route. Essentially it shows the success path traversed by the routing algorithm. For example in the snapshot below, we can see it tried to match the DefaultApi route, failed, skipped the next one that excludes axds and finally hit the successful route. If you have seen Phil Haack's Route Debugger, this feature is similar.



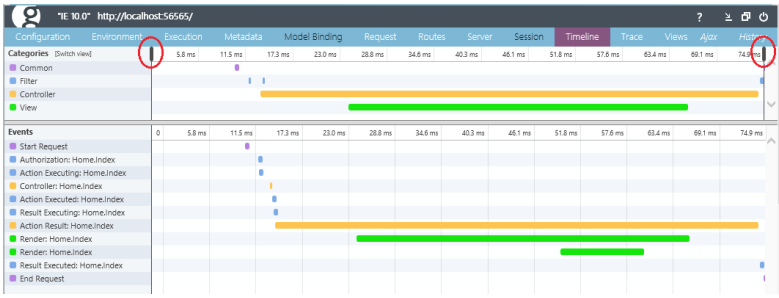
Timeline

The timeline feature breaks down the time required for the request to complete. There are actually two views, one a hierarchical view as follows



Second view is a flattened out view. The Categories that are color coded can be unselected to show a filtered list.

What's more interesting is that you have two markers that can be moved to select a smaller cross section of the categories to zoom-in.

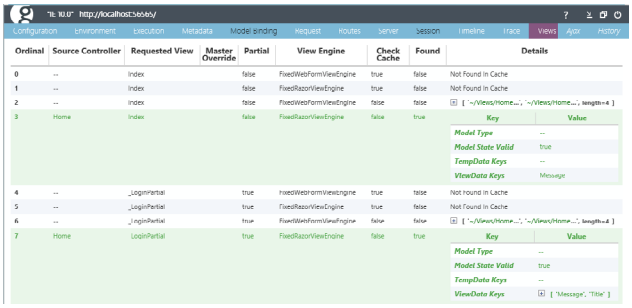


Views

This is probably the most insightful view for someone new to MVC. This shows how MVC selects a particular View. For a single request that has one partial view, we see how

MVC queries the WebFormsViewEngine first and then the RazorViewEngine. It first searches the cache and if it doesn't find it, the cache requests the view from the Disk.

Essentially, the MVC view resolution code walks through all the View engines asking them if they were going to serve up the requested view. As is the theme for all the tabs, the successful hit is highlighted in green.

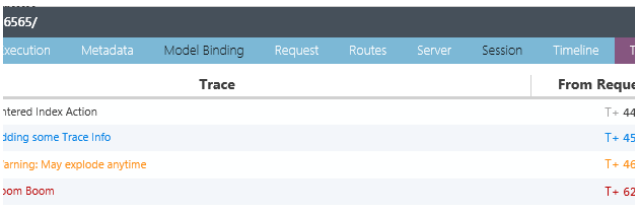


Trace

Finally we come to the Trace tab. This where custom tracing information is displayed. Custom as in explicit Trace statements that you may have written in the application. For example, if we update the Home Controller's Index action method with the following trace statements:

```
public ActionResult Index() {
 System.Diagnostics.Trace.Write("Entered Index
Action", "HomeController");
 System.Diagnostics.Trace.TraceInformation("Adding
some Trace Info", "HomeController");
 System.Diagnostics.Trace.TraceWarning("Warning: May
explode anytime");
 ViewBag.Message = "Modify this template to jump-
start your ASP.NET MVC application.";
 System.Diagnostics.Trace.TraceError("Boom Boom");
 return View();
}
```

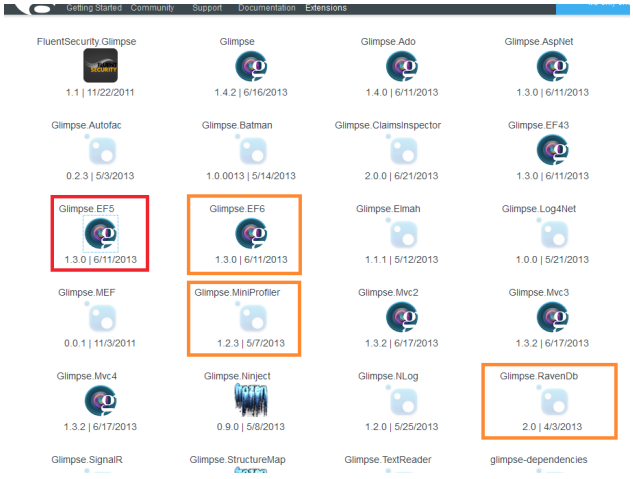
The Trace Tab looks as follows:



So essentially all custom tracing info that we put is displayed in this tab, with helpful color coding based on the type of Trace.

THE GLIMPSE ECO-SYSTEM

Now that we have got a whiff of what Glimpse does, we must be wondering umm, what about tracing DB calls? For such an elaborate profiling system, surely there's a way to database calls? Well, this is a good time to introduce you to the Glimpse Eco-System



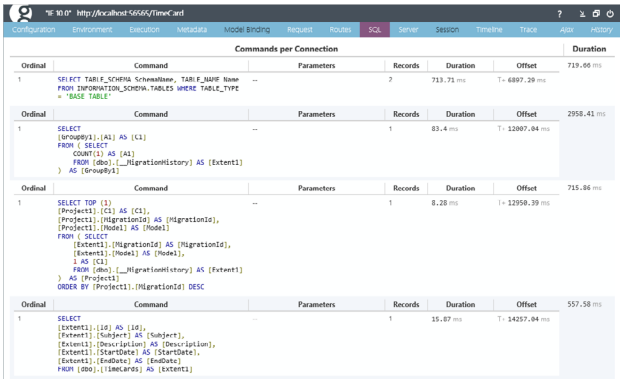
Well this is only a partial list of the Extensions available. As we can see we have EntityFramework, RavenDB, heck we even have an extension for Miniprofiler.

Profiling Entity Framework Database Calls

To get started with Database Profiling, when you use EntityFramework in your Data Layer, install the correct version of the EF extension. In our case, I installed EF5 from the Nuget Package Manager Console as follows:

```
PM> Install-Package Glimpse.EF5
```

Once installed we run the application and navigate to TimeCard index page. We see that a new SQL tab has been added to the Glimpse panel. The following set of queries surprised me.



As you can see, there are two additional queries that I didn't anticipate, one to the INFORMATION_SCHEMA and two to the __MigrationHistory table. These three are strictly EF related, and happen only on first run of the application. If you refresh the page, only the call to TimeCards table will be registered.

SECURITY AND PROFILES IN GLIMPSE

The information that Glimpse displays is potentially sensitive and should be 'exposed' very carefully. Inadvertent exposure of data like Connnection strings etc. can be disastrous from a security perspective. However, Glimpse is safe by default. The Web.config's Glimpse section is by default defined as follows:

```
<glimpse defaultRuntimePolicy="On"
endpointBaseUri="~/Glimpse.axd">
  <!-- Want to use Glimpse on a remote server?
Ignore the LocalPolicy by removing this comment.
    <runtimePolicies>
      <ignoredTypes>
        <add type="Glimpse.AspNet.Policy.LocalPolicy,
        Glimpse.AspNet"/>
      </ignoredTypes>
    </runtimePolicies>-->
</glimpse>
```

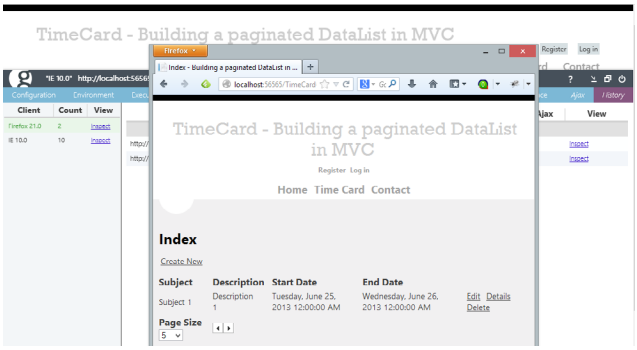
The defaultRuntimePolicy="On" sets up Glimpse to work locally ONLY. To disable this we have to add an <ignore-Types> (commented out above) and add the Glimpse.AspNet. Policy.LocalPolicy assembly name.

Custom Policies

However if we want to customize our policies, we could create a Custom Policy file as well. Glimpse very helpfully adds a commented out Class called GlimpseSecurityPolicy. The Execute method in this class receives the HTTP context and we can use the context to determine if Glimpse should be turned on or off for the given request. One example specified is to check if the given User is a part of the Administrator group and if not, turn off Glimpse.

OTHER FEATURES

Glimpse has this neat feature where it can track requesting clients. So if you have enabled Glimpse for multiple clients, you can sit at one browser and monitor the activities of the other browser. This can be ideally used for remote debugging. Combined with a Custom Policy, we could create a 'Debugger' group in our ACL and add users who are having problems to the Debugger group. Then request them to reproduce the error at their end while you monitor the information thrown up by Glimpse for that user. In the screenshot below, we have the hypothetical scenario where the second user is on a different machine using Firefox.



On our browser in IE, we click on the History tab to see there are two clients connected. When we click on Firefox, the data displayed switches to Firefox and we can see what's going on at the other client.

That completes our round up of the hottest features of Glimpse. At this point, you should be sufficiently equipped to start poking around and inspecting your app with Glimpse.

CONCLUSION

Glimpse is truly an awesome addition to an ASP.NET developer's toolbox. It provides deep insights into the working of an ASP.NET application without too many invasive code changes. Thanks to Redgate's sponsorship, the project is placed well with project leaders devoting 100% of time to it. Over the last two years, it has got impressive community support.

Today we primarily looked at the 'out-of-the-box' features. Someday in future, we can investigate the extension points and how to add our own extensions to Glimpse ■

Download the entire source code from our GitHub Repository at bit.ly/dncm7-gli

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

SCRIPTCS - C#, THE WAY YOU WANTED IT TO BE

Filip W. the co-coordinator of the ScriptCS project, a Microsoft MVP, introduces us to this cool new OSS project. It's an exciting way to use C#, so exciting that Sumit (our editor) couldn't help himself and jumped in with a sample himself.

ScriptCS is an open source initiative, aimed at bringing a revolutionary, friction-less scripting experience to C# developers.

If you have ever longed the node.js or JavaScript style development experience when developing C# applications, then ScriptCS is something exactly for you.

Its goal is to provide a lightweight, yet robust, approach to authoring C# programs from outside Visual Studio - in any text editor, without having to worry about a solution or project file, build tasks, top level classes, assembly references or namespace imports. Just like you would expect from a modern language with rapid iteration (write-refresh-check) development cycle.

On top of all that, ScriptCS integrates tightly into Nuget, and provides its own extensibility model called "script packs", based on the node.js "require" concept.

ScriptCS also ships with its own REPL - read–eval–print loop - giving you a command line shell where you can write and execute C#, and use all ScriptCS features (including script packs).

Under the hood, ScriptCS uses Roslyn CTP as the execution engine. It is however abstracted away, and work is under way to create a ScriptCS engine.

GETTING STARTED

The easiest way to get started with ScriptCS is to install it from Chocolatey (http://chocolatey.org/), which is a Machine Package Manager, somewhat like apt-get, but built with Windows in mind.

```
cinst ScriptCS
```

This will pull the latest stable version of ScriptCS and add it into your environment's PATH variable. If you wish to play with the latest versions, you can install ScriptCS from our nighlty MyGet feed:

```
cinst ScriptCS -pre -source https://
www.myget.org/F/scriptcsnightly/
```

If you are feeling a little less adventurous, you could try the stable Nuget feed and go with just

```
cinst ScriptCS
```

Once installed, simply close the command window and reopen it.

WRITING YOUR FIRST SCRIPT

To start developing with ScriptCS, simply pull up your favorite text editor (if you use Sublime Text, we have a special ScriptCS plugin which includes code highlighting) and start writing in a *csx file.

The simplest program would look like this (suppose we name it start.csx):

```
Console.WriteLine("Hello World");
```

Notice there is no namespace, no class, no static void method or no using statements. You can execute this program from command line:

```
ScriptCS start.csx
```

```
D:\_dnc>scriptcs start.csx
INFO : Starting to create execution components
INFO : Starting execution
Hello World
INFO : Finished execution

D:\_dnc>
```

By default the following namespaces are included for you (explicit using statements for them aren't needed):

- System
- System.Collections.Generic
- System.Linq
- System.Text
- System.Threading.Tasks
- System.IO

If you wish to use classes from other namespaces, you simply bring them in with a *using* on top of your file.

Also, ScriptCS allows you to define functions (methods) like you would in JavaScript - without a need to wrap in a class.

```
void SayHello()
{
    Console.WriteLine("Hello!");
}

SayHello();
```

```
D:\_dnc>scriptcs start.csx
INFO : Starting to create execution components
INFO : Starting execution
Hello!
INFO : Finished execution

D:\_dnc>
```

REFERENCING OTHER SCRIPTS AND DLLS

You can load other script files from your main script file. This is done through a #load preprocessor directive.

For example, consider other_script.csx file:

```
var x = 1;
Console.WriteLine("Hello from other script!");
```

And a main script:

```
#load "other_script.csx"

Console.WriteLine("Hello World from Main script!");
Console.WriteLine("Variable from other script: {0}",
x);
```

```
D:\_dnc>scriptcs start.csx
INFO : Starting to create execution components
INFO : Starting execution
Hello from other script!
Hello World from Main script!
Variable from other script: 1
INFO : Finished execution

D:\_dnc>
```

Variables, classes, functions defined in referenced ScriptCS are available in the context of the script that references them. Evaluation statements (such as the Console.WriteLine) are processed immediately.

#load has to be used at the top of your CSX file - that is, before the first line of "normal" C# code.

You can also reference traditional DLLs in your script. This is done through the #r syntax.

You can either reference a DLL from a GAC, i.e.:

```
#r "System.Web"

var address = HttpUtility.UrlEncode("€€€");
```

Or you can drop any DLL into a bin folder relative to your script and reference that:

```
#r "My.dll"

var obj = new MyTypeFromMyDll();
```

USING NUGET WITH SCRIPTCS

ScriptCS is very well integrated with Nuget - and can install packages based on packages.config or through command line, without the need of nuget.exe.

To install a package you can simply type:

```
ScriptCS -install <package_name>
```

```
D:\_dnc>scriptcs -install microsoft.aspnet.webapi.selfhost
INFO : Installing packages...
INFO : Installed: microsoft.aspnet.webapi.selfhost
INFO : Installation completed successfully.
INFO : Copying assemblies to bin folder...
INFO : Copied: System.Net.Http.Formatting.dll.
INFO : Copied: System.Web.Http.dll.
INFO : Copied: System.Web.Http.SelfHost.dll.
INFO : Copied: System.Net.Http.dll.
INFO : Copied: System.Net.Http.WebRequest.dll.
INFO : Copied: Newtonsoft.Json.dll.
INFO : Restore completed successfully.
Initiated saving packages into packages.config...
Added Microsoft.AspNet.WebApi.Client, Version 4.0.30506.0, .NET 4.0
Added Microsoft.AspNet.WebApi.Core, Version 4.0.20710.0, .NET 4.0
Added Microsoft.AspNet.WebApi.SelfHost, Version 4.0.20918.0, .NET 4.0
Added Microsoft.Net.Http, Version 2.0.20710.0, .NET 4.5
Added Newtonsoft.Json, Version 4.5.11, .NET 4.0
Packages.config successfully created!

D:\_dnc>
```

At the end of this type of installation, ScriptCS will create a packages.config file which you can distribute with your script (without having to copy the DLLs). Which brings us to another possible installation type - if you have a packages.config and place it in the same folder as your script, you can just do:

```
ScriptCS –install
```

which will simply install everything based on that packages file.

If you use nightly builds, you don't even have to perform an explicit installation - if you run a script and have a packages.config in your folder and no packages have been installed yet, ScriptCS will automatically install them as you run the script for the first time.

For more information on interacting with Nuget, I recommend you have a look at the detailed article in the ScriptCS wiki at https://github.com/ScriptCS/ScriptCS/wiki/Package-installation.

EXTENDING SCRIPTCS WITH SCRIPT PACKS

You can streamline many of the development activities when working with common libraries by using ScriptCS script packs. They are also installed from Nuget, and they are responsible for importing additional namespaces, referencing additional namespaces and providing shortcuts into pre-configured objects that can be accessed in your script context.

All the script packs are available off the script host through an ambient, global require method:

```
Require
```

For example the following command will install a NancyFx script pack (https://github.com/adamralph/ScriptCS-nancy).

```
ScriptCS -install ScriptCs.Nancy
```

In your script you can now do:

```
Require<NancyPack>().Host();

public class SampleModule : NancyModule
{
    public SampleModule()
    {
        Get["/"] = _ => "Hello World!";
    }
}
```

This is an extraordinarily small amount of code (1), but it creates and starts a fully functional self-hosted Nancy service for you on port 8888 (note, the use of HTTP listener here requires the command line to be running with elevation).

```
INFO : Starting to create execution components
INFO : Starting execution
Searching assembly: R*60e3b12a-2d12-48ea-bc4b-78c3874ba008-#1UD, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Searching assembly: ScriptCs.Nancy, Version=0.6.0.0, Culture=neutral, PublicKeyToken=null
Found Nancy module: Submission#0+SampleModule
Hosting Nancy at: http://localhost:8888/
Press any key to end
```

We already have a nicely growing ecosystem of script packs created by different community contributors - including script packs for ASP.NET Web API, ServiceStack, WPF, Azure Mobile Services, CLR diagnostics, NUnit and many more. You can find an up to date master list of script packs at https://github.com/ScriptCS/ScriptCS/wiki/Script-Packs-master-list.

USING THE REPL

Whatever you can do in the script, you can also do in our REPL. To launch REPL, simply start ScriptCS without any parameters:

```
ScriptCS
```

This will fire the interactive console, and you can start writing C# code. ScriptCS uses ServiceStack.Text to serialize and output the statements that return a value. This way you don't have to use Console.Write to output the values you want to see.

```
D:\_dnc>scriptcs
scriptcs (ctrl-c or blank to exit)

> var y = 100;
> y
100
>
```

In REPL mode, you can also use #load and #r as you would in normal script. If you recall our previous example, start.csx:

```
D:\_dnc>scriptcs
scriptcs (ctrl-c or blank to exit)

> #load "start.csx"
Hello from other script!
Hello World from Main script!
Variable from other script: 1
>
```

EDITOR'S NOTE – BATCH PROCESSING USING SCRIPTCS

Encouraged by Filip's introduction, our Editor Sumit Maitra decided to dive deeper into ScriptCS. The idea of using C# as a scripting language was intriguing indeed.

Given that Windows now comes with a powerful scripting shell - the PowerShell, ScriptCS at first chance seems to duplicate efforts. But PowerShell can be intimidating at first. If you are more comfortable with C# than PowerShell scripting, ScriptCS seemed like a perfectly valid alternative given that you could do pretty much everything C# (in this case Roslyn) can do. With this use case in mind, I decided to explore it further.

An Example

Say we have our music library and would like to get a list of duplicates. It's pretty easy to classify duplicates based on the file size. So we could script it up and run it from the command line. Using ScriptCS, we wouldn't have to worry about Visual Studio or solution type etc. We could simply save the script in a CSX file and run it. Our script would have to do the following:

1. Not worry about including/importing namespaces. We don't really have to refer to any new namespace because System.IO is already a part of the default libraries that are loaded.

2. Declare a generic Dictionary of long keys and List<string>. This will have all the files found keyed by their size. If there are multiple files of the same size, their names will go in the List<string>.

3. Use the ScriptArgs object to grab arguments from the command line. The ScriptArgs collection is populated with command line arguments that we pass while running ScriptCS. However to segregate arguments meant for ScriptCS runtime and arguments meant for the script, we use the -- separator. So if we type the following command

```
C:\> ScriptCS HelloWorld.csx -- -arg01 val01
```

Only the arguments after the -- will be passed to ScriptArgs, i.e. –arg01 and val01

So we look for the parameter --scan and check if the Directory name passed actually exists. If it does, we call the ScanFolders method. This recursively checks the current folder and all the subfolders for files with extension mp3.

4. The SaveName method saves files names in the dictionary keyed by Size.

5. Once all the files have been parsed recursively, we loop through each item in the dictionary. If any of the lists contain more than one file name, it implies they could be duplicates and thus print their names out.

The Script

The complete listing of the Script is as follows:

```
Dictionary<long, List<string>> files = new
Dictionary<long, List<string>>();
Console.WriteLine("Args: {0}", string.Join("','",
ScriptArgs));

if(ScriptArgs[0] == "-scan")
{
 if(Directory.Exists(ScriptArgs[1]))
 {
  Console.WriteLine("Scanning {0}", ScriptArgs[1]);
  ScanFolders(new System.IO.DirectoryInfo(
   ScriptArgs[1]));
  Console.WriteLine("\n\n******** Scanning {0} COMPLETE
********", ScriptArgs[1]);
  foreach(var v in files)
  {
   if(v.Value.Count > 1)
   {
    Console.WriteLine("Following {0} Files have the
     same size \n -'{1}'", v.Value.Count, string.
     Join("',\n -'", v.Value));
   }
  }
 }
}

void ScanFolders(System.IO.DirectoryInfo dirInfo)
{
 try
 {
  IEnumerable<System.IO.FileInfo> files =
   dirInfo.EnumerateFiles("*.mp3");
  Parallel.ForEach<FileInfo>(files, SaveName);
```

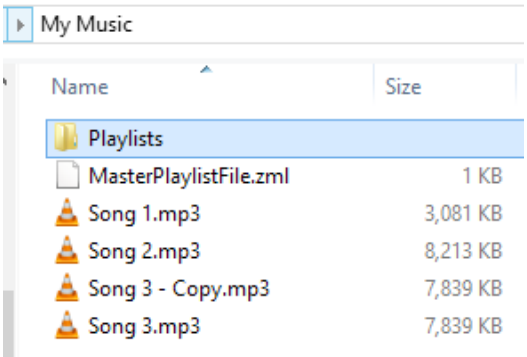
```
  IEnumerable<DirectoryInfo> directories =
   dirInfo.EnumerateDirectories();
  if (directories.Count<DirectoryInfo>() > 0)
  {
   Parallel.ForEach<DirectoryInfo>(directories,
   ScanFolder);
  }
 }
 catch(Exception ex)
 {
  Console.WriteLine("Borked: {0}", ex.Message);
 }
}

void ScanFolder(System.IO.DirectoryInfo currentFolder,
ParallelLoopState arg2, long arg3)
{
 ScanFolders(currentFolder);
}

void SaveName(FileInfo currentFileInfo,
ParallelLoopState arg2, long arg3)
{
 if(files.ContainsKey(currentFileInfo.Length))
 {
  files[currentFileInfo.Length].Add(
   currentFileInfo.FullName);
 }
 else
 {
  files[currentFileInfo.Length] = new List<string>();
  files[currentFileInfo.Length].Add(
    currentFileInfo.FullName);
 }
}
```

Running the Sample

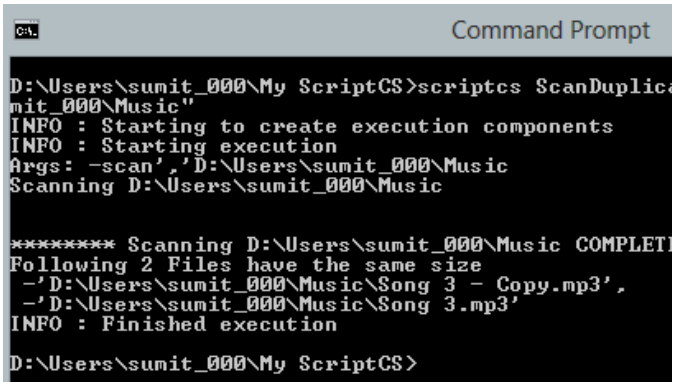
We start off the following test data set.



As we can see, Song 3.mp3 is duplicate. We run the script as follows

```
D:\Users\sumit_000\My ScriptCS>scriptcs ScanDuplicates.
csx -- -scan "D:\Users\sumit_000\Music"
```

This gives us the following output.



Sweet, our script has found the duplicates for us. In future, we can enhance it by writing the result to a file, introducing a flag to delete the duplicate files so on and so forth.

Though scripting C# sounds a little strange to start off with, but by the time I was done with the above script, the ease of saving the script and directly running it on the console was already growing on me and I am definitely a fan of ScriptCS now.

Given the wonderful community support it is gathering, people will find increasingly more productive ways to use ScriptCS. I personally foresee a lot of automation stuff in future using ScriptCS. Given enough support, who knows, it might just become a parallel scripting shell in its own rights!

FURTHER READING

These are the basics of ScriptCS - and hopefully enough to get you up and running! There is far more to ScriptCS than just that. I have listed some helpful resources on our Github page & it would be great to see you one day become a ScriptCS contributor! ■

Download the entire source code from our Github repository at bit.ly/dncm-scs

Filip is the co-coordinator of the ScriptCS project, a Microsoft MVP, popular ASP.NET blogger, open source contributor. He is a member of the ASP.NET Web API advisory board and blogs at www.strathweb.com. You can find him on Twitter at @fillip_woj

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

DNC MAGAZINE

(IF YOU HAVEN'T ALREADY)

01 It's free!!! Can't get anymore economical than that!

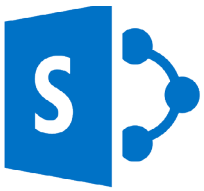
02 Every issue has something totally new from the .NET world!

03 The magazines are really well done and the layouts are a visual treat!

04 The concepts are explained just right, neither spoon-fed nor too abstract!

05 Through the Interviews, I've learnt more about my favorite techies than by following them on Twitter

Subscribeat
www.dotnetcurry.com/magazine



SharePoint 2013

BUSINESS CONNECTIVITY SERVICES AND ODATA

Pravinkumar Dabade shows us how to use OData to perform CRUD operations on External Data via Business Connectivity Services in Sharepoint 2013

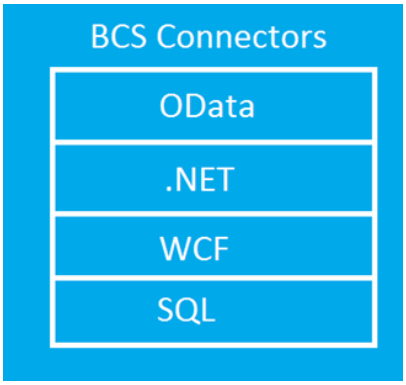
Before we start with Business Connectivity Services in SharePoint Server 2013, let's look back at SharePoint Server 2007 and SharePoint 2010 for some background. SP 2007 had the concept of Business Data Catalog (BDC). BDCs allowed us to integrate Line of Business systems' data into SharePoint Server 2007. However, working with BDC was a little tedious due to its complex XML implementation.

In SharePoint Server 2010, Microsoft Business Connectivity Services made developer's life simpler by adding support for connectors like Database, .NET Assemblies and WCF Services. We could now Read and Write an LOB system's data without too much customization and only by using Microsoft SharePoint Designer 2010/2012. In case of complex data retrieval logic, we could make use of .NET Assemblies or WCF Services and in such cases, use Visual Studio 2010/2012.

BCS in SP 2010 provided external data integration directly into SharePoint as well as Office clients, either online or offline. When working with BCS, we could perform Create, Retrieve, Update and Delete Operations. We could use Web Parts, Lists, Microsoft Outlook Tasks and Appointments for displaying external data accessed through BCS.

One of the best features of BCS was performing operations on external offline data. For example Microsoft Outlook 2010, Microsoft SharePoint Workspace. When data was manipulated offline, it got synchronized when the connection was reestablished.

With all these features already in place, Microsoft SharePoint Server 2013 introduced an additional BCS connector – OData. This gives us four out-of-the-box BCS Connectors – OData, .NET dlls, WCF as well as SQL.



BUSINESS CONNECTIVITY SERVICES AND OTHER CORE CONCEPTS

Before we jump into using OData with BCS, let's quickly go through some of the core BCS concepts that were established in SharePoint 2010.

External Content Types

ECT enables reusability of business entity metadata. It also provides built-in behavior for Microsoft Office Outlook items such as Contacts, Tasks, Calendars, Microsoft Word Documents and SharePoint lists and web parts.

By using ECT, the information workers need not worry about how complex the External Systems are or how to connect with these systems or their locations. ECT follows the security of the external system as well as SharePoint, thus allowing us to control the data access by configuring the security under SharePoint.

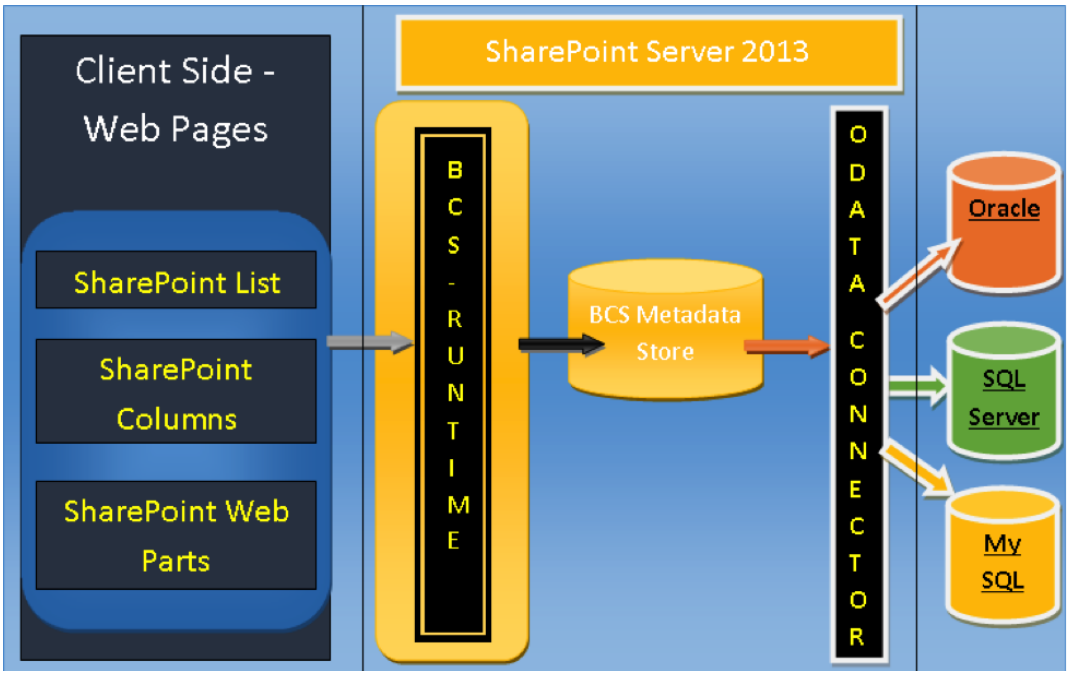
We can use External Content Type in SharePoint as a source of the external data as below -

1. External Lists in SharePoint which uses ECT as a data source.
2. External Data Column uses ECT as data source to add the data to Standard SharePoint Lists.
3. SharePoint Business Connectivity Services offers various web parts like -
   (a) External Data List web part.
   (b) External Data Item web part.
   (c) External Data Related List web part.

OData and BCS

Now let's talk about OData. OData (Open Data Protocol) is a web protocol for performing CRUD operations built upon web technologies like HTTP, Atom Publishing Protocol (AtomPub) and JSON to provide access to the data to various applications, services and stores. In SharePoint 2013, Business Connectivity Services can communicate with OData sources. SharePoint list data is exposed as an OData source hence SharePoint is already a producer of OData Source.

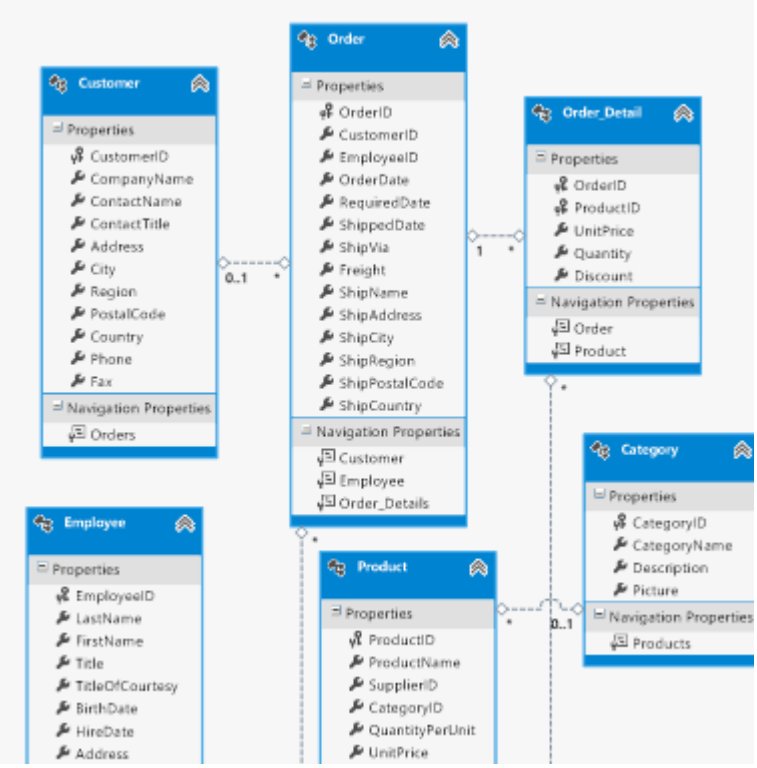
Below is the architecture diagram of how SharePoint makes use of OData for fetching external system data.



The diagram shows how a SharePoint web page that contains a Web Part or SharePoint Column or a SharePoint list, can access the data from the external data sources. Users can manipulate the data and perform operations like CRUD (Create, Retrieve, Update and Delete) if the user has permissions and ECT supports these operations.

BCS Runtime is responsible for taking a request for accessing the external data. Then it makes a connection with the external data source using the configured connector.

The BCS runtime accesses the BDC Metadata store and retrieves the ECT, which is stored in SQL Server. This database only contains information about the connections for external data sources. Once the BDC runtime gets the information about the External Content Type and connection information, it passes the request to the external system with the help of connector.



The connectors get connected with the data source and perform the necessary operations and return data back to the user.

FETCHING DATA USING ODATA

Now let's design an OData service which will fetch the data from a sample like the Northwind database in SQL Server.

To create an OData service, open Visual Studio 2012 and create an Empty Web Application with the name "NorthwindODataService". Then right click the web project and add a new Item. From the Visual C# section located at left side, select "Data" and choose ADO.NET Entity Data Model. Click on "Add" button. Now it will show you an Entity Data Model Wizard. Select Generate from database. Configure the connection string to your Northwind database and click on "Next" button.

In this step, we will choose the entities. Expand the Tables section and from dbo schema choose "Customers", "Orders", "Order Details", "Employees" and "Products". Once you choose the tables, click on the Finish button. Your Entity Model should look similar to the following –

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

```
e NorthwindODataService

ic class NWService : DataService<NorthwindEntities>

public static void InitializeService(DataServiceConfiguratio
{
    config.SetEntitySetAccessRule("*", EntitySetRights.All);
    // config.SetServiceOperationAccessRule("MyServiceOperat
perationRights.All);
    config.DataServiceBehavior.MaxProtocolVersion = DataServ
}
```

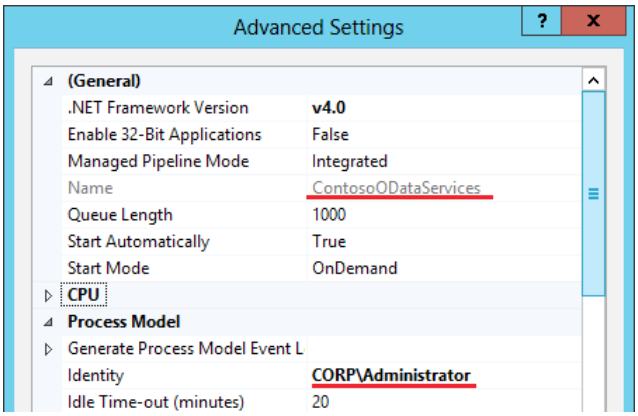
Once the service is configured, you can test the service. Right click the NWService.svc file and click on "View in Browser". You will see all the Entities which we have chosen. Try changing the URL as below –

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

You can now try executing various queries against your OData service. It's time to deploy your service under IIS. You can deploy your OData service in various ways. I am going to deploy the OData service using manual deployment.

Open Internet Information Services (IIS) Manager. Create a blank folder under C:\inetpub\wwwroot\OData folder with the name OData. Create a new web site with the name DNCODataServices and map the above OData folder to the Web site.

Then copy all the required files from NorthwindODataService web project and paste those files under "C:\inetpub\wwwroot\OData". Now the most important thing is to set the Identity of the Application Pool which got created when we created our Web site. It should look like the following –



Now test the OData service by using different queries using URIs which we have used last time for testing our Northwind OData Service.

Now that our OData service is ready, let's use it in our SharePoint Server 2013. As discussed above, we can use Business Connectivity Services to fetch the External data into SharePoint. To fetch the External Data into SharePoint, we will now design a BDC Model which will make use of our OData service as a data source to fetch the data from Northwind database. This returns a big blob of XML that we will not reproduce here, but will look at more closely in parts.

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

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

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

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

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

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

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

g. Read Specific Customer. The URL is –

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

h. Create Customer. The URL is –

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

i. Update Customer. The URL is –

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

j. Delete Customer. The URL is –

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

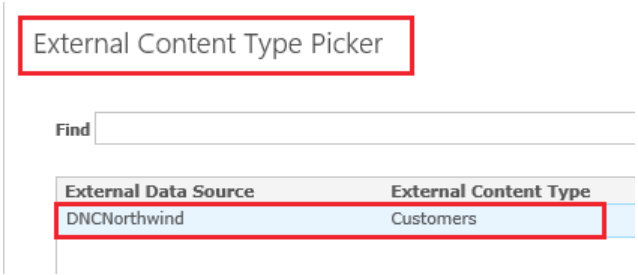
Once the Model is ready, let's import it into our Business Connectivity Services using SharePoint Central Administration Tool. To import the model, open Central Administration and click on Application Management. Under Application Management, Service Applications, click on Manage Service

Applications.

Click on "Business Data Connectivity Service". On the "Edit" ribbon, click on "Import" button and browse the BDC Model file. This will import the file into Business Data Connectivity Service. The most important part is set the permissions to the entities of our BDC Model. Once the permissions are set, the model is ready to use.

Open SharePoint Site if it already exists. If you don't have one, create a new site collection using Team Site Template.

Now let's add a new list in the site which is "External List" with the name DNC Customers. Click on the "Select External Content Type" button. It will show you External Content Type Picker window. Select DNCNorthwind -> Customers as shown below –

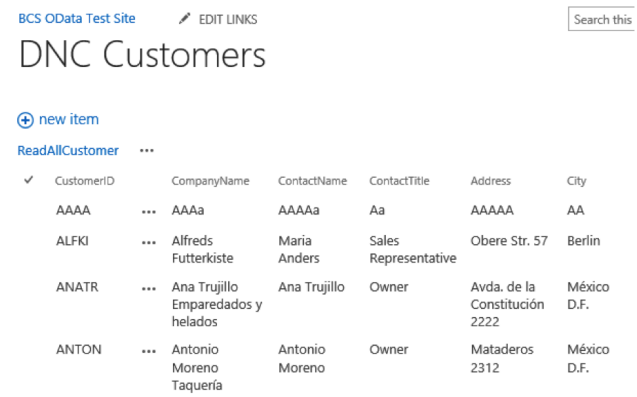


Click on Create button which will create a new list. Click on the newly created list and you will see the Northwind Customers' table records under SharePoint External List.

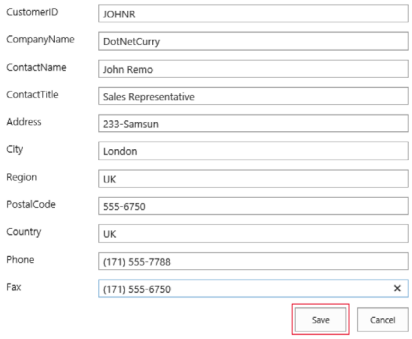
DEMO

Now let's test our ECT by performing some CRUD operations -

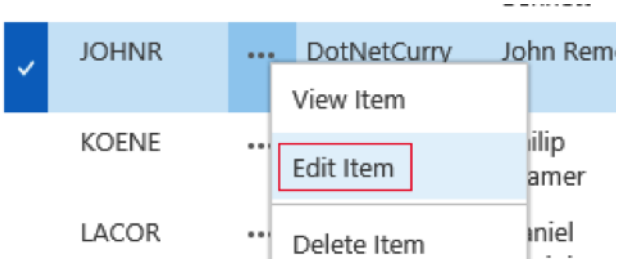
1) Navigate to the newly created List and we'll see the data is read from Customers table –



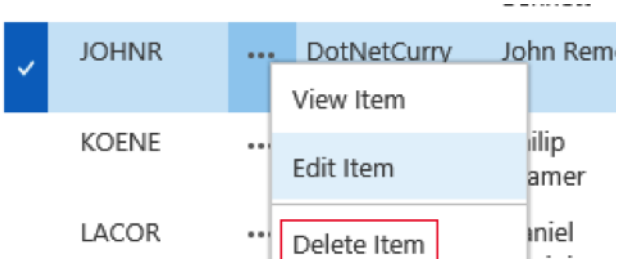
2) Now let's add a record – Click on the 'new item' button shown in the image we just saw



3) To update the same record click on 'Edit Item' as shown below –



4) Similarly you can delete the item as well–



That concludes our Demo.

CONCLUSION

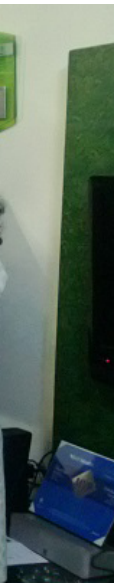
To conclude, we saw how we could leverage Business Connectivity Services in SharePoint 2013 and communicate with OData sources. Addition of the OData connector adds another level of flexibility to BCS and makes it even easier to use SharePoint as your central information management portal ■

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


Join us on
Facebook
www.facebook.com/
dotnetcurry

PICGALLERYDNCMAGAZINEANNI

VERSARY**PARTY**2013



01 "Every Episode is a New Adventure" - Sumit and Suprotim share their 'DNC Mag Go Live Adventures'

02 Everyone listening to Subodh Sohoni share his Consulting battle stories

03 Yummy! Happy Birthday #DNCMag

04 "Joke's on you bro" - Subodh and Gouri Sohoni crack a joke with Mahesh and Sumit

05 "Someone please cut the cake!"

06 "How many geek devs does it take to cut a Cake?" - Well we all had a hand in this!

07 Okay, Congratulations - One last time!

08 The Wolf Pack! From LtoR: Praveen, Mahesh, Subodh, Suprotim, Minal, Gouri and Sumit

MICROSOFT PEX FRAMEWORK AND ITS NEW COUSIN CODE DIGGER

Raj Aththanayake walks us through an interesting MS Research project called Microsoft Pex. In this article we will look at some of the features of Microsoft Pex in Visual Studio 2010. We also look at the VS 2012 plugin Code Digger that integrates Pex in VS 2012

Unit Testing has been around for a long time and today it has become an important aspect of Software Development. Unit Testing has evolved over the years and there has been significant research into improving both tooling and techniques around Unit Testing.

Microsoft Pex is one of the research tools that is slowly but steadily getting popular in the .NET world. Pex stands for "Program Exploration" and it makes Unit Testing lot easier and fun than ever before.

Microsoft Pex has adapted the concept of White Box testing, which is basically testing an application or function based on its internal working structure. The internal structure can be categorized in number of ways such as Branches, Paths, Data flow etc.

GETTING STARTED WITH MICROSOFT PEX

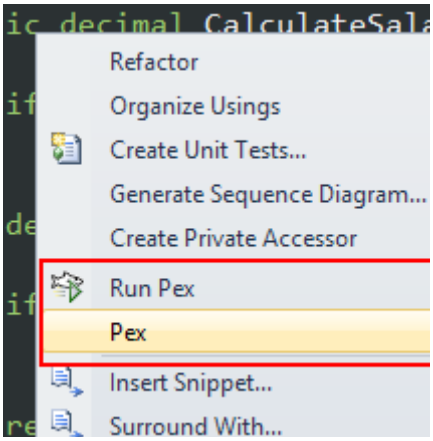
You can download Pex from Microsoft Research Pex website.

Note that Pex also comes with a tool called Moles, which is now replaced by the powerful VS 2012 Fakes framework. At the time of writing Pex is not supported by the new VS 2012. It has been replaced by a tool called **Code Digger** which is a lightweight version of Pex. I will describe Code Digger in more details in a bit.

Pex can be used from VS 2005 onwards. However the full VS integration is only available from VS 2008. The Pex examples that have been demonstrated in this article use VS 2010.

The main idea behind Pex is based on Parameterized Unit Testing. We will have a look at this in more details in - **"Parameterized Unit Testing with Pex".** For now, it is important to understand that Pex helps us to auto generate Unit Test cases based on the structure of the program. This means that you no longer need to write hand written Unit Tests to extensively exercise a method, as Pex framework is capable of auto generating a significant set of test cases for you.

Once you have installed Pex in VS2010, you can see the Pex commands in the context menu as below.



DOES PEX COMPLETELY REPLACE HAND WRITTEN UNIT TESTS?

The short answer is No. In practice, I find that Pex is really good for generating various test cases for functions that are predominantly based on many branches that can take various range inputs as parameters. A good example would be an algorithm that you want to extensively test various inputs in. For these types of algorithms, hand written Unit Tests are exhaustive to write and have no guarantee that we cover all the required test cases.

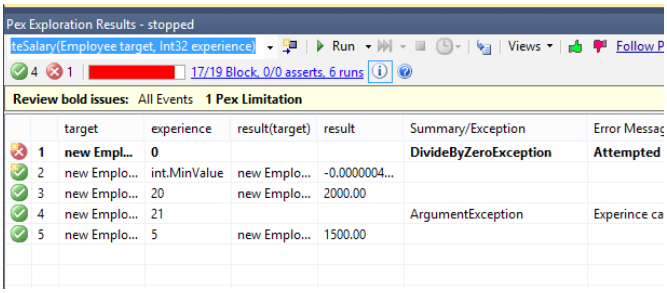
In most other cases, you still require hand written Unit Tests. For example, you want to be sure that your components are well designed as you would normally do with practices such as TDD (Test Driven Development)

GENERATING UNIT TESTS WITH PEX

Let's have a look at a C# function, and generate the test case for this function using Pex. We have the following sample function to calculate Salary

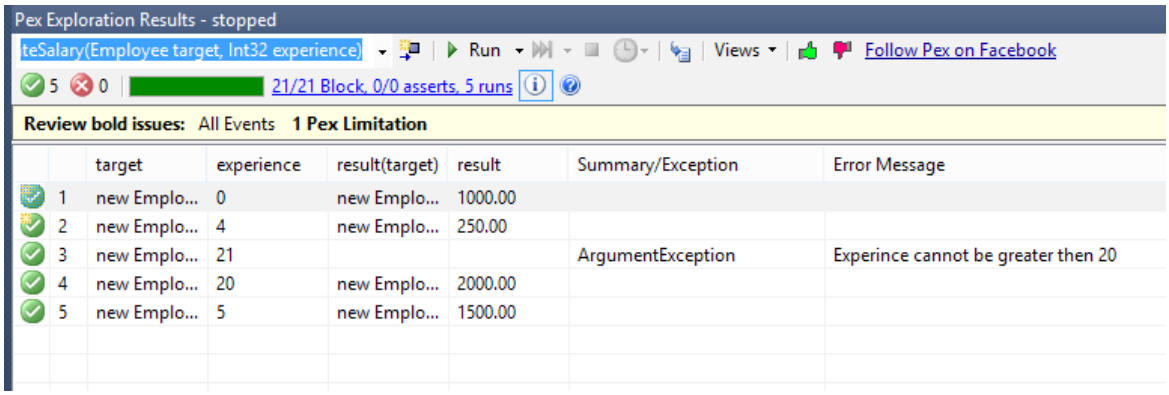
```
public class Employee
{
 public decimal CalculateSalary(int experience)
 {
  if (experience > 20)
  {
   throw new ArgumentException("Experience cannot be
                               greater than 20");
  }
  decimal baseSalary = 1000.00m;
  if (experience < 5)
  {
   return baseSalary / experience;
  }
  if (experience >= 5 && experience <= 10)
  {
   return baseSalary + 500.00m;
  }
  if (experience > 10)
  {
   return baseSalary + 1000.00m;
  }
  return baseSalary;
 }
}
```

In the menu after clicking "Run Pex", you can see Pex has generated the inputs and outputs as below.



Pex also generates inputs that can potentially make the program to throw unhandled exceptions, such as DivideByZeroException. The good thing about this is that you can look at the failing test and modify your program, so it can handle DivideByZero exceptions.

```
public decimal CalculateSalary(int experience)
{
 if (experience > 20)
  throw new ArgumentException("Experince cannot be
greater then 20");
 decimal baseSalary = 1000.00m;
 if (experience < 5 && experience > 0)
  return baseSalary / experience;
 if (experience >= 5 && experience <= 10)
  return baseSalary + 500.00m;
 if (experience > 10)
  return baseSalary + 1000.00m;
 return baseSalary; }
```



How Pex generates inputs and outputs

The Pex engine looks at the method's every branch, statement, and data. It then generates inputs and outputs accordingly. The technique called Dynamic Symbolic Execution is used to create these inputs and outputs. As you see, Pex feeds the simplest possible values first to a method and incrementally it generates inputs for corner cases.

If you were to add another condition, Pex would generate another entry in the table. This means that you would get really high code coverage for your routine.

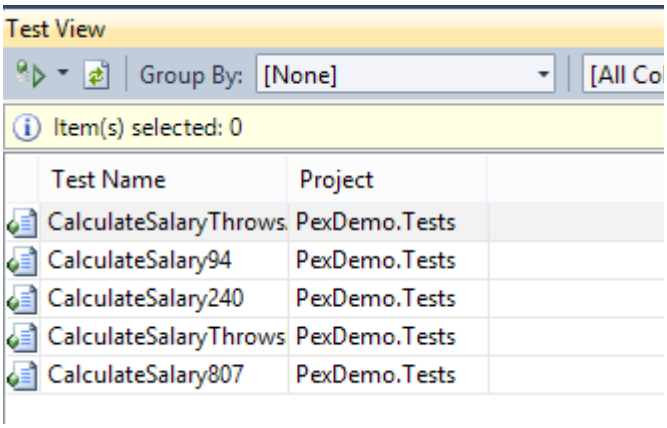
Pex generates all these Unit Tests separately in a code file which you have access to.
See some examples below.

```
[TestMethod]
[PexGeneratedBy(typeof(EmployeeTest))]
[ExpectedException(typeof(ArgumentException))]
public void CalculateSalaryThrowsArgumentException38()
{
 decimal d;
 Employee s0 = new Employee();
 d = this.CalculateSalary(s0, 21);
}

[TestMethod]
[PexGeneratedBy(typeof(EmployeeTest))]
public void CalculateSalary94()
{
 decimal d;
 Employee s0 = new Employee();
 d = this.CalculateSalary(s0, 20);
 Assert.AreEqual<decimal>(200000e-2M, d);
 Assert.IsNotNull((object)s0);
}
```

This also means that if you cannot understand any input and how the code produces a corresponding output/result, you can easily debug the generated tests. (Right click on any test and select "Debug").

All Pex generated Unit Tests can also be run separately from standard VS test tool window.



Each of the above generated test calls a Pex parameterized test method as below.

```
/// <summary>This class contains parameterized unit
tests for Employee</summary>
[PexClass(typeof(Employee))]
[PexAllowedExceptionFromTypeUnderTest(typeof(
InvalidOperationException))]
[PexAllowedExceptionFromTypeUnderTest(typeof(
ArgumentException), AcceptExceptionSubtypes = true)]
[TestClass]
public partial class EmployeeTest
{
 /// <summary>Test stub for CalculateSalary(Int32)
 /// </summary>
 [PexMethod]
 public decimal CalculateSalary([PexAssumeUnderTest]
  Employee target, int experience)
 {
```

```
  decimal result = target.CalculateSalary(experience);
  return result;
  // TODO: add assertions to method EmployeeTest.
  CalculateSalary(Employee, Int32)
 }
}
```

Does Pex support other test frameworks?

Pex does not support other test frameworks out of the box. However you can download Pex Extension from Codeplex, that allows you to use Pex with your own test framework http://pex.codeplex.com/

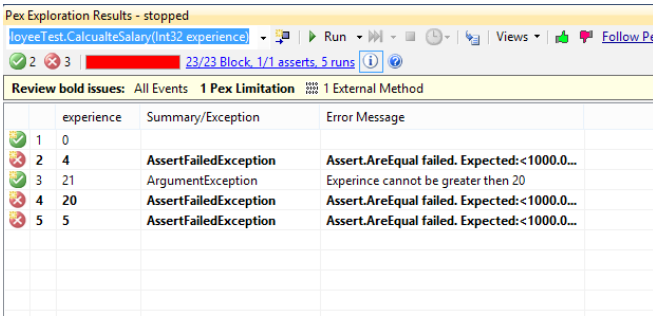
PARAMETERIZED UNIT TESTING WITH PEX

One of the neat features of Pex is that you can create parameterized Unit Tests to customize your Tests. Once you specify a parameter in your Unit Test method as argument, Pex auto generates inputs (based on the branches and the statements), calls the system under test and then asserts the return value. To make parameterized Unit Test, you must decorate the Unit Test method with [PexMethod] attribute.

```
[PexMethod]
public void CalculateSalary(int experience)
{
 var sut = new Employee();
 var result = sut.CalculateSalary(experience);
 Assert.AreEqual(1000.00m, result);
}
```

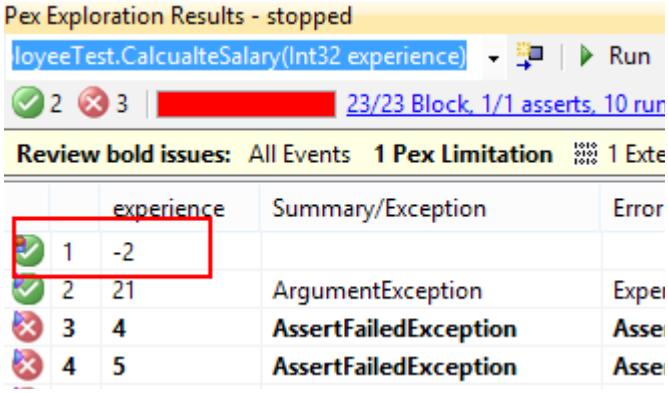
Note that generating Parameterized tests can also be achieved by right clicking on the class under test, *selecting Pex > Create ParamaterizedUnitTests*

Once you create your own parameterized test method, right click on the test method and select "Run Pex Explorations". You should be able to see the Pex generated inputs and outputs Assert against the Unit Test's assert condition.



If you want additional customization against the parameters that has been passed into the Unit Test, you can use the [PexArguments] attribute as below. This will generate additional inputs and outputs based on the arguments specified by the [PexArguments] attribute.

```
[PexMethod]
[PexArguments(-2)]
public void CalcualteSalary(int experience)
{
 var sut = new Employee();
 var result = sut.CalculateSalary(experience);
 Assert.AreEqual(1000.00m, result);
}
```



You can easily debug any failing tests using standard Visual Studio debug features.

MICROSOFT CODE DIGGER

Code Digger is a lightweight version of Pex that specifically targets VS 2012 Portable class libraries. At this stage, Microsoft Pex, which we looked at, is not compatible with VS 2012 yet. I believe the future version of Code Digger may replace Pex and would target other project types.

You can download the Code Digger Extension from VS Extension Gallery

Once you have installed the Code Digger plugin (VS Professional and above), you should be able to see a new menu

item “Generate Inputs/Outputs Table” in the context menu.

Let’s say we have a simple C# method that checks specific names within a list.

```
public bool ListContainNames(List<string> listNames)
{
 if (listNames == null) return false;
 if (listNames.Any())
 {
  if (listNames.ElementAt(0).Contains(“smith”))
  {
   throw new Exception(“should not contain smith”);
  }
  if (listNames.ElementAt(0).Contains(“david”))
  {
   return true;
  }
 }
 return false;
}
```

After running Code Digger, you should be able to see a table that generates inputs and outputs very similar to Pex generated table which we saw earlier.

Inputs / Outputs - stopped

target	listNames	result(target)	result	Summary / Exception	Error Message
new CodeDiggerDemo()	null	new CodeDiggerDemo()	false		
new CodeDiggerDemo()	{}	new CodeDiggerDemo()	false		
new CodeDiggerDemo()	**{null}**			**NullReferenceException**	**Object reference not set to an instance of a**
new CodeDiggerDemo()	{""}	new CodeDiggerDemo()	false		
new CodeDiggerDemo()	{"\0\0\0\0\0"}	new CodeDiggerDemo()	false		
new CodeDiggerDemo()	{"david"}	new CodeDiggerDemo()	true		
new CodeDiggerDemo()	**{"smith"}**			**Exception**	**should not contain smith**

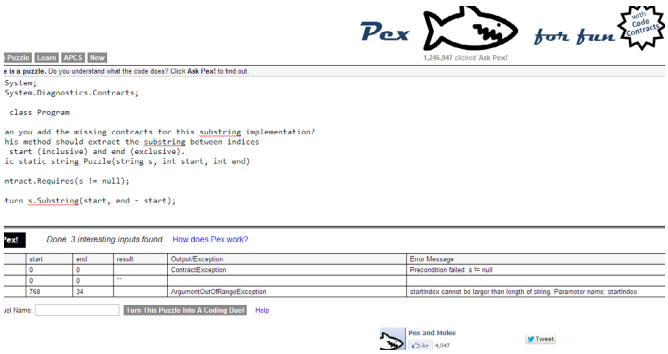
Note: Unlike Pex, at this stage Code Digger does not have any auto generated Unit Tests or parameterized Unit Tests.

PEX FOR FUN (PEX4FUN.COM)

What we saw so far is automatic test case generation integrated within Visual Studio. Pex4Fun is a web site that uses the same Pex engine in the server side but provides more interactive or gaming style experience to the user. It is a simple interface, and you post C# code in the text editor and click “Ask Pex”. Pex sends the code to the server and validates the code, generates test data and sends back the result (see screenshot to the right)

If you copy and paste this code in VS, you would see pretty much the same test cases generated by Visual Studio PEX engine.

The idea of this web site is to provide puzzles and Coding Duels so it can gamify Unit Testing. Once you have selected a puzzle,



and after generating inputs and outputs, your task is to modify the program so the Pex no longer produces any failed results. Once you have no failed results, you have solved the puzzle. It is fun and you can use this program to solve various puzzles.

SUMMARY

As you know, these are research tools and some have not completely made it into the final/latest Visual Studio. If you are using VS2010, you can use Pex extensively and generate a boiler plate regression test suite. Code Digger seems to be using the same Pex engine for test generation and we would expect substantial changes around automated testing in upcoming VS releases. It is very promising and lot developers would benefit from.

I recommend use Pex for complex algorithms, to get a leg up in generating maximum possible test cases automatically. Once Pex is done, we can review the generated test cases and augment them with hand-written Unit Tests. There are still great benefits of writing your hand written Unit Tests (TDD or Non-TDD).

Pex4Fun is a great web site that allows you to solve coding puzzles. ■

Downloaded the entire source code from our GitHub repository at bit.ly/dncm7-pex

Raj Aththanayake is a Microsoft ASP.NET Web Developer specialized in Agile Development practices such Test Driven Development (TDD) and Unit Testing. He is also passionate about technologies such as ASP.NET MVC. He regularly presents at community user groups and conferences. Raj also writes articles in his blog http://blog.rajsoftware.com. You can follow Raj on twitter @raj_kba


Follow us on
Twitter
@dotnetcurry

OWIN, KATANA & SIGNALR – BREAKING FREE FROM YOUR WEB SERVER

The deployment story of Web based Services (like WCF and Web API) in the .NET world has traditionally been tied to IIS. The Open Web Interface for .NET (OWIN) standard defines a standard interface between .NET web servers and web applications. This aims to break server and application components down into smaller and simpler modules.

Sumit Maitra demonstrates OWIN and how we can host a SignalR based application using Katana (an OWIN implementation).

OWIN stands for Open Web Interface for .NET. Now if you are thinking, .NET already has ASP.NET and MVC and Web API, how does this OWIN thing fit in? Is it a new tool/library/web framework, and then what is Katana?

Well OWIN is an open, industry standard or a specification. For now let's assume it is a specification that outlines 'some' rules for building 'certain' types of .NET applications. We'll come back to the types shortly.

The Katana project by Microsoft picks up these rules and has created an implementation of the specification. So OWIN is like the blueprint whereas the Katana project uses the blueprint to create increasingly useful set of modules that adhere to the OWIN rules. Don't worry about what those Modules do, we'll see that soon, for now, imprint this much, OWIN is a blueprint and Katana in as implementation.

WHY DO WE NEED OWIN?

With the vague idea of what is OWIN and Katana, let's go deeper and see what is the grand scheme of things in which OWIN and Katana fit in. After all what is the OWIN specification for?

The Microsoft Web Application Stack today

We first take a step back and see how Web Applications work on ASP.NET today from a hosting point of view. Any webdev, worth their salt knows that during development, you use either Cassini, the development web server, or IIS Express, the newer more robust, in process web server. But when we deploy our ASP.NET apps, it's invariably on an IIS server. IIS is the be-all-end-all of ASP.NET Web Hosting at least from a production deployment stand point. With

Some OWIN Facts

OWIN which stands for Open Web Interface is primarily authored by two members of the ASP.NET Team - Benjamin Vanderveen and Louis Dejardin.

You can compare OWIN with RACK in Ruby and WSGI from the Python world

> *OWIN is a specification of interoperability between web hosts and frameworks without any dependencies between them*

IIS we know that we will get a set of extension points, but we will get things like Compression, Authentication, Static content serving etc. for free. All this making IIS one giant black box into which we immerse our ASP.NET application. We can visualize it as follows.

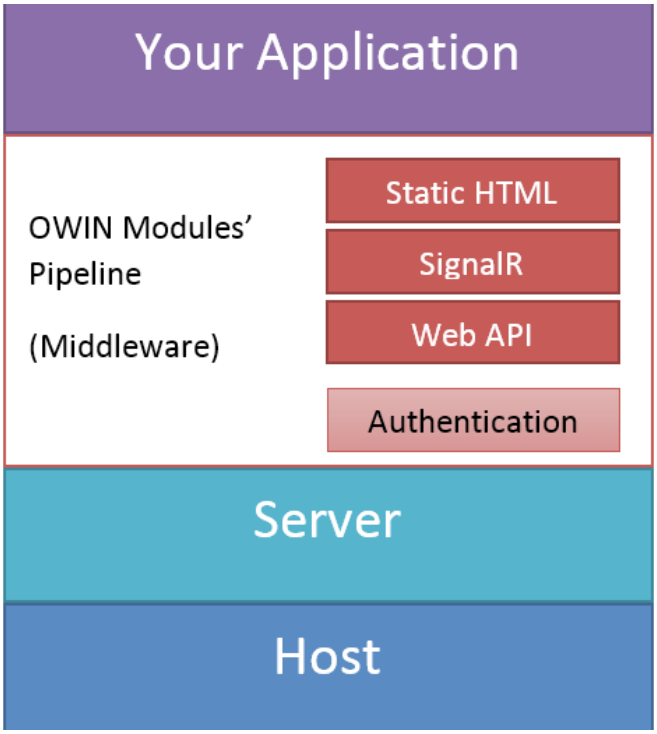
This looks like a pretty picture till we start thinking of using our application in scenarios without IIS.

Do such scenarios exist?

Well, recently we reviewed the Sitefinity CMS application and their demo was making innovative use of Cassini, they could have very well used an alternative if available. Think of application stacks like Pentaho and Jasper Reports in the Java world. They don't depend on a preconfigured web server to work, they come bundled with a Web Server which could be Apache or could be something smaller and lighter.
So yes, scenarios do very much exist!

THE GRAND SCHEME OF OWIN AND THE KATANA PROJECT

OWIN aims to change the picture as follows



The OWIN spec splits the entire stack into four layers

1. Host
2. Server
3. Middleware
4. Custom Application

Katana as we briefly mentioned above is a suite being built, that will have its own OWIN compliant implementations of the Host, Server and Middleware layers. The difference (from IIS) being it will be far more granular giving you way more control on how to structure your application. The final goal being not to replace IIS but provide a suite of functionality that can be 'assembled' to comprise of only the pieces that you need for your application.

Thus the above diagram can also be considered as the 'architecture' for Katana. Now let's look at each of these components in details.

Host

Hosts are near the OS, low level pieces of functionality that do things like

- Manage the underlying system processes
- Manage the server selection, lining up the OWIN modules and handling of requests.

Katana (the OWIN implementation from Microsoft) has (planned) support for three Hosting options.

a. IIS/ASP.NET: Via HttpModule and HttpHandler implementations, Katana manages to implement the OWIN pipeline requirements and execute OWIN compliant modules. This is managed by installing the Microsoft.AspNet.Host.SystemWeb Nuget package in any Web Application that you may be working on. There is one caveat though, IIS acts as both the Host and the Server, so when you use the Nuget package, you have to run with this 'limitation' and cannot swap in another Server implementation

b. Custom Host: This is a closer to the metal option that allows users to use a Console application, Windows Process or even a WinForms application to be the host. We will look at this option in a lot more details.

c. OwinHost.exe: This is another 'implementation' coming out of the Katana suite. This provides a little more support in terms of bootstrapping by providing a Console Host that starts an Http Server on its own so that you can focus on configuring the middleware and building the final application.

With the hosting sorted out, let's move to the next layer that Katana has in store which is the Server layer.

Server

Often the distinction of Host and Server is merged and the fact that the OWIN spec recognizes these two as separate entities, speaks to the granularity and pluggability OWIN is aiming to achieve. As a base implementation the Katana project provides two Server implementation.

a. Microsoft.Owin.Host.SystemWeb: As mentioned above, the IIS/ASP.NET host module serves as both the Host and Server implementation under IIS/ASP.NET. As a result, it's quoted in both the categories.

b. Microsoft.Owin.Host.HttpListener: This is a more barebones implementation using the .NET Frameworks' HttpListener class. It contains the plumbing required to open ports and channel requests to the requests in the OWIN pipeline/middleware. We will use this in our sample below.

This brings us to the crux or if we may say, Strength of the OWIN spec, the Middleware!

Middleware/OWIN Modules' pipeline

As depicted in the diagram, the pipeline of OWIN modules basically channels a request from the application to the server. This pipeline approach greatly modularizes the overall solution giving the end user ability to pick and choose components required. As we will see shortly, if we are building a static HTML site, then we need not include SignalR, WebAPI or Authentication. On the other hand, if we are building an MVC application, we can use framework like Fubu MVC or Nancy and ignore Web API as well as Microsoft's MVC stack completely. Essentially, flexibility of having an HttpHost outside IIS opens up a plethora of permutations and combinations that are today possible only if you agree to bring along the overhead of a full blown Server like IIS.

At a very basic level, an OWIN Middleware Component must implement a Func that takes in an IDictionary or objects and returns a Task.

```
Func<IDictionary<string, object>, Task>
```

Once you have a middleware component, you can wire them up as follows

```
public class Startup
{
 public void Configuration(IAppBuilder app)
 {
  app.Use(<type of middleware object>, <parameters that
it needs>);
 }
}
```

Note the class name Startup and the method Configuration and its signature. This forms a convention for Katana Hosts.

Each middleware tends to implement its own set of Extension methods for Use, however for a Host and Server agnostic implementation, the above method signature must be implemented correctly.

Once all required middleware has been registered, we can then build Applications on top of this stack. In the next section, we will see how we can build an application using Katana OWIN components.

BUILDING A SIGNALR APP USING KATANA OWIN COMPONENTS

SignalR's self-host story is built on Katana. The SignalR team went ahead and built a Server and the Middleware for themselves.

Today we'll build a Self-Hosted Console app using Katana components that not only hosts SignalR server, but also static HTML pages. In the end, we'll have a crude chat application that uses Signal, HTML and JavaScript only. No WebForms or ASP.NET MVC involved.

The Console Host App

For the first part, we pretty much follow the instructions on SignalR's Wiki, however as we will see, we go beyond the barebones instructions.

Step 1: We start off with a Windows Console Application and call it – SignalRSelfHost

Step 2: We pull in the dependencies required to build a self-hosting SignalR using Nuget

1. First up we get Owin Hosting: This has the WebApplication class that provides us the bootstrapping necessary to do ' self-hosting'.

```
PM> Install-Package Microsoft.Owin.Hosting –pre
```

2. Next we get the Owin HttpListener

```
PM> Install-Package Microsoft.Owin.Host.HttpListener –
pre
```

3. Finally we get SignalR's Owin host implementation

```
PM> Install-Package Microsoft.AspNet.SignalR.Owin
```

Step 3: In the main method, add the following code to initialize our Host. We set the default url to localhost:9999. If there is a –url command line argument passed to the console application, we try to use that as the url for our self-hosted server.

Next we send the url and the Type Startup to the WebApplication class' Start method. This starts off the Self host Server. If your Visual Studio is not running in Administrator mode, you may get an exception here because of lack of port opening permissions. Just start Visual Studio in Administrator mode.

Note: Startup is a naming convention that OwinHosts follow. We'll see the Startup class shortly.

So finally our Console app code is as follows:

```
public class SignalRSelfHost
{
 static string url = "http://localhost:9999";
 static void Main(string[] args)
 {
  if (args.Contains<string>("-url"))
  {
   url = GetUrl(args);
  }
  using (WebApplication.Start<Startup>(url))
  {
   Console.WriteLine("Server running on {0}", url);
   Console.ReadLine();
  }
 }
 private static string GetUrl(string[] args)
 {
  for (int i = 0; i < args.Length; i++)
```

```
 {
   if (args[i] == "-url" && args.Length > i+1)
   {
    return args[i + 1];
   }
  }
  return url;
 }
}
```

Step 4: Next we add a new class called Startup.cs

The class name Startup and the method Configuration with an IAppBuilder input parameter as we saw is a convention that Katana looks out for, and uses to configure the MiddleWare modules.

In our implementation below, we see that we have created a (SignalR) HubConfiguration object with Cross Domain enabled and used the MapHubs extension method to pass the configuration to instantiate and pass the SignalR Hub middleware into the OWIN chain.

```
public class Startup
{
 public void Configuration(IAppBuilder app)
 {
  // Turn cross domain on
  var config = new HubConfiguration { EnableCrossDomain
= true };
  // This will map out to http://localhost:9999/signalr
  app.MapHubs(config);
 }
}
```

At this point, we have SignalR hooked up but it doesn't have a Hub associated with it. So let's add a SignalR Hub next.

Step 5: We add a class called MyHub that has one function SendMessage and all it does is, lobs back the message it receives to all connected clients.

```
public class MyHub : Hub{
 public void Send(string message)
 {
  Clients.All.addMessage(message);
 }
}
```

At this point, we have SignalR middleware and SignalR Hub ready to go. Only thing missing are clients. Just out of curiosity, if you run the application and navigate to the [server url]/signalr/hubs your browser will download a JavaScript file for you. If we open the JS file, we'll see it's the JavaScript hub proxy and a proxy for our MyHub.Send method in it.

But how do we add an HTML/JavaScript client? We could create a new ASP.NET application, but that would kind of defeat the whole purpose of the self-hosting effort. What if we could host static HTMLs along with the SignalR middleware, and build our clients in pure JavaScript?

This is where we realize the beauty of OWIN. We can actually use a Katana module that allows us to host plain HTML. Let's see how.

Adding Static HTML Hosting to our Self-Hosted Application

Note: We are dealing with the 'bleeding edge' here as in most of the Katana Owin components are Beta or Alpha release. The Static HTML Hosting is an alpha release. So expect some tweaks when things go live late in 2013.

Step 1: We install Static HTML Hosting middleware module using Nuget as follows

```
PM> Install-Package Microsoft.Owin.StaticFiles –version
0.20-alpha-20220-88
```

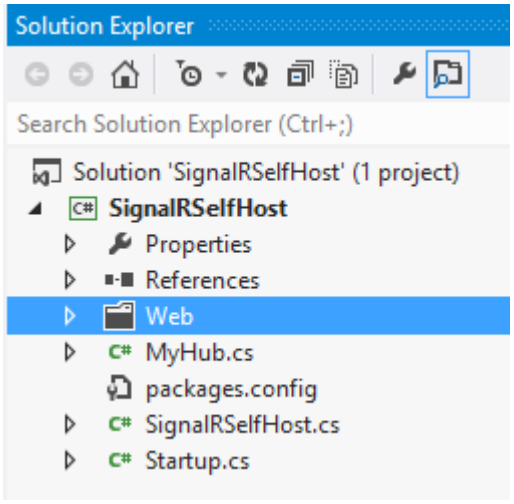
It required some creative reverse engineering to get the StaticFiles package off Nuget.

Step 2: Once installed we head back to the Startup class and add the highlighted code below

```
public class Startup
{
 public void Configuration(IAppBuilder app)
 {
  var config = new HubConfiguration { EnableCrossDomain
= true };
  app.MapHubs(config);
  string exeFolder = Path.GetDirectoryName(Assembly.
GetExecutingAssembly().Location);
  string webFolder = Path.Combine(exeFolder, "Web");
  app.UseStaticFiles(webFolder);
 }
}
```

This basically tells the Static Files host, that it should look for a folder called Web at the executable's location and use all files under it for hosting.

Step 3: Create a folder named Web in the solution



Step 4: In the Web Folder, add a new HTML file called Home.html. This will be our web client but before we implement anything here, we need to get the SignalR dependencies in first.

Step 5: Let's install the SignalR client side scripts from Nuget

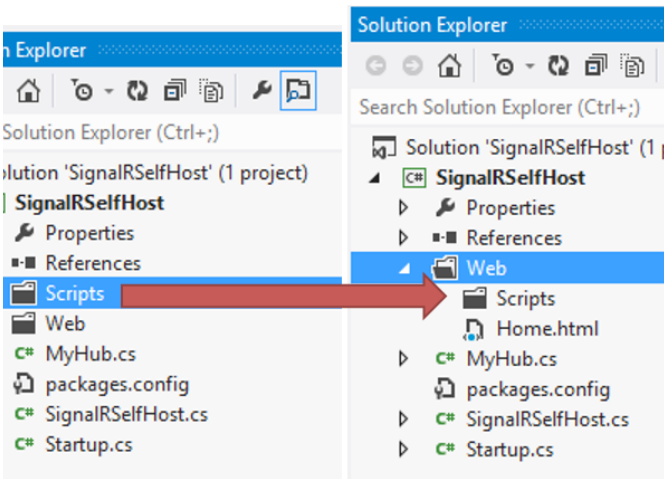
```
PM> install-package Microsoft.AspNet.SignalR.JS
```

Notice it will get jQuery 1.6.x. This is rather old, so we'll upgrade jQuery to latest using the following command

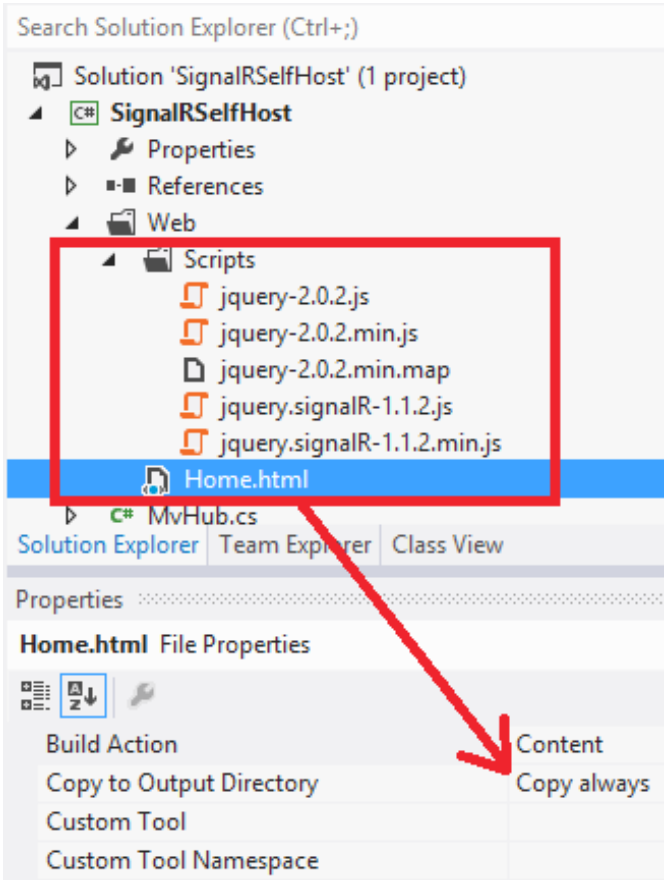
```
PM> update-package jquery
```

This will install jQuery and SignalR clients to the Scripts folder at the Solution's root.

Once installed, move the Scripts folder under the Web folder as follows:



Step 6: Select each file under Web folder (or its subfolders) and change the "Copy To Output Directory" property to 'Copy Always". This ensures all our changes are updated on each run.



IMPLEMENTING THE SIGNALR CLIENT

Our plumbing for Static hosting is now complete. All that we have to do is implement a SignalR client. Let's do a rudimentary message exchange sample. We will have two Text Boxes, one for incoming messages and one for outgoing messages. We'll have one button to send the text, in the outgoing message textbox, to all connected clients. It isn't very pretty as we can see below:

The markup for it is as follows:

```
<!DOCTYPE html>
<html>
 <head>
  <title>SignalR Self Host</title>
   <!-- jQuery, SignalR scripts -->
   <script src="../signalr/hubs"></script>
```

```
 </head>
 <body>
  <h1>SignalR Self Hosting</h1>
  <div>
   <div>
    <textarea id="messagesText" rows="20"
     style="width:100%">
    </textarea>
   </div>
   <div>
    <textarea id="newMessage" rows="3"></textarea>
    <button id="sendMessage">Send</button>
   </div>
  </div>
  <script type="text/javascript">
   <!-- We'll see the script below -->
  </script>
 </body>
</html>
```

It is rather un-remarkable except for the Magic script source of "../signalr/hubs". This is the dynamic proxy that is generated at runtime, which maps to the server side hub.

The SignalR Implementation

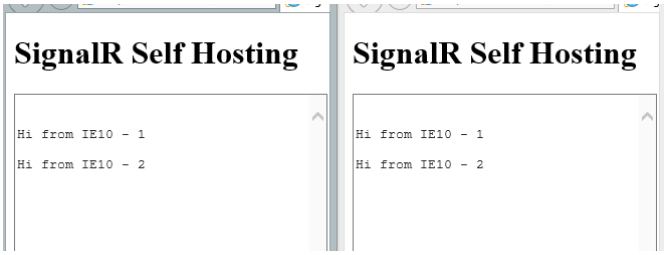
The implementation is very simple:

- We retrieve an instance of the client side hub proxy.
- Add the addMessage method to the hub's client so that it can be invoked by the server
- Next we initialize the client connection. Once initialization is done (async), we assign an event handler for the button click. The handler function uses the hub's server proxy to invoke the 'Send' method in our Hub. The Hub, on receipt lobs it back to all clients and the addMessage method is invoked on all of them. The complete script is as follows:

```
<script type="text/javascript">
  var hub = $.connection.myHub,
  $msgText = $("#messagesText"),
  $newMessage = $("#newMessage");
  hub.client.addMessage = function (message)
  {
    $msgText.text($msgText.text() + "\r\n" + message);
  }

  $.connection.hub.start().done(function () {
  $(document).on('click', '#sendMessage', function(){
  hub.server.send($newMessage.text());
  });
  });
</script>
```

And we are done!!! Let's build the application and open a couple of browser instances. I have two IE10 browser instances open side by side. As I type in one and hit send, the message comes up in both the browsers simultaneously. Sweet!



CAVEATS

As I mentioned earlier, the Self hosting bits are pre-release as is the static hosting. I had some issues where sometimes the connections would be refused specially when using Firefox, so I dropped it from the demo. But these are all going to be fixed by the final release towards the end of this year.

CONCLUSION

This concludes our OWIN and Katana introduction with a walkthrough using SignalR and Static hosting middleware in the pipeline. We saw the granularity of control that we have when introducing features in our application's stack. Just like we added SignalR and the Static hosting, we could also add Web API and then call Web API controllers from HTML pages using AJAX. Similarly if we put an Authentication module in front of the pipeline, we could put all requests behind an Authentication layer. However the biggest strength of the OWIN implementation is independence from IIS stack and granularity of control. Another aspect that we didn't cover is the increase in testability, thanks to less monolithic pieces of infrastructure ■

Download the entire source code from our GitHub repository at bit.ly/dncm7-swin

Sumit is a .NET consultant and has been working on Microsoft Technologies for the past 12 years. He edits, he codes and he manages content when at work. C# is his first love, but he is often seen flirting with Java and Objective C. You can Follow him on twitter @sumitkm and read his articles at bit.ly/KZ8Zxb

SQL AZURE

MIGRATING ON-PREMISES DATABASE TO CLOUD AND USING SQL REPORTING

Windows Azure is a cloud based platform that can be leveraged for deploying and managing enterprise applications in a public cloud or a hybrid cloud setup. This can often reduce cost of managing applications when compared to On-premises application deployment.

One of the challenges here is migrating a database from an on-premises database server to the cloud. Windows Azure SQL Database extends SQL Server capabilities to the cloud. Advantages of having the database on the cloud includes high availability, scalability and familiar development model.

In this article, we will look at how we can leverage Windows Azure SQL Database and Reporting Services to move an application to the cloud.

KEY BENEFITS OF WINDOWS AZURE SQL DATABASE

Let's look into the key benefits of the Windows Azure SQL Database:

• **High Availability** - Windows Azure SQL database is built on Windows Server and SQL Server technologies. Once the database is created, multiple copies of the data are replicated across multiple physical servers. This helps in high availability of the data for Business solutions like Finance.

In case of any hardware failure, automatic failover ensures that data availability is not hindered.

• **Scalability** - This is the real reason why we choose database over cloud. Windows Azure SQL provides automatic scaling as the data grows. Of course this is a paid

"SQL Azure provides high availability, scalablity and familiar development experience amongst other features."

model. The cost depends upon the subscription you purchased.

• **Familiar development experience** - Developers can continue to use their current programming models (e.g. ADO. NET, Entity Framework etc).

• **Relational Data Model** - As a database developer, your familiarity with T-SQL, Stored Procedures, Views, etc. can be utilized while working with the Windows Azure SQL database.

In the rest of the article, we will see how to:

- Do Data Migration and Backup.
- SQL Reporting on SQL Azure Database.

WORKING WITH DATA MIGRATION

When you have an application to be ported to the cloud, and if this application is making use of on-Premises SQL Server database, then instead of creating database from scratch, you can use Windows Azure SQL database to migrate your existing on-premises database (schema and data) to Azure. To perform the migration, we have the following tools available to us:

- Import/Export Service.
- Generate and Publish Scripts Wizard.
- The BCP Utility.

Along with the above mentioned tools, you can also get a free tool SQL Server Migration Wizard from CodePlex. In this article, we will see the use of Import/Export service.

Using Import/Export Service

This functionality was introduced in SQL Server Management Studio 2012 to migrate on-premises databases to SQL Azure. This service creates a logical backup file i.e. BACPAC file. This file contains schema definition and table data of the database.

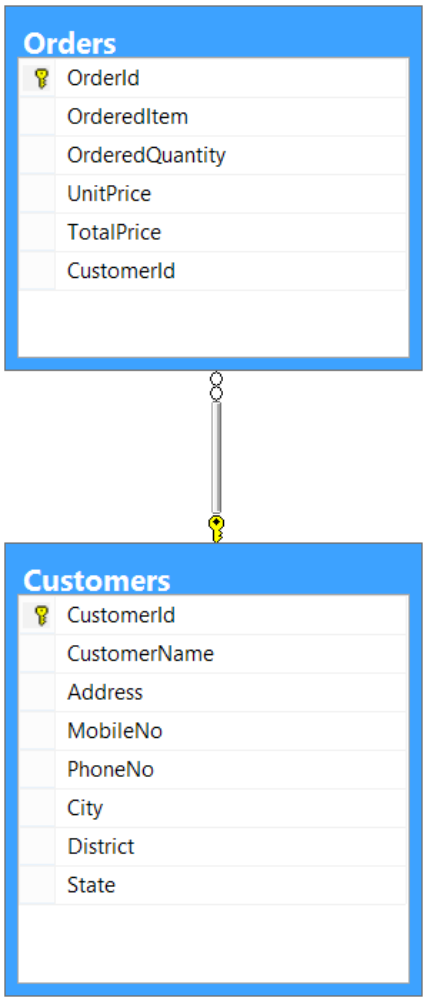
The concept of the BACPAC, was originally introduced by the Data-Tier Application Framework (DAC Fx) which is set of tools, APIs and services designed to improve development, deployment and management of SQL Server

database schemas. We first saw DAC Fx with SQL Server 2008 R2. Now in SQL Server 2012, DAC Fx Version 2.0 is available with the ability to export to BACPAC.

Task 1: Deploying directly from On-Premises SQL Server to SQL Azure

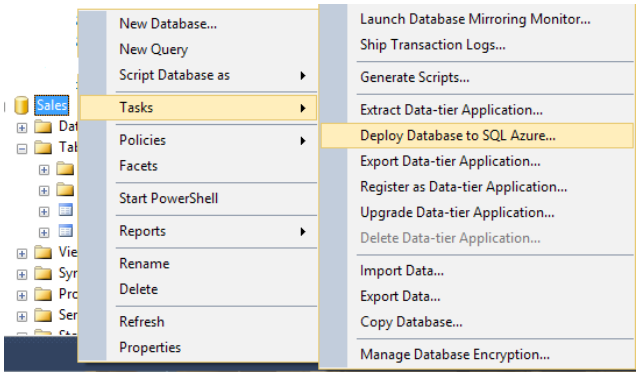
We will first use the easiest mechanism to migrate our on-premises SQL database to SQL Azure. In SQL Server 2012, using DAC Fx, deploying on-premises database to SQL Azure directly is now possible.

Step 1: Consider the Sales Database with the following Schema tables:

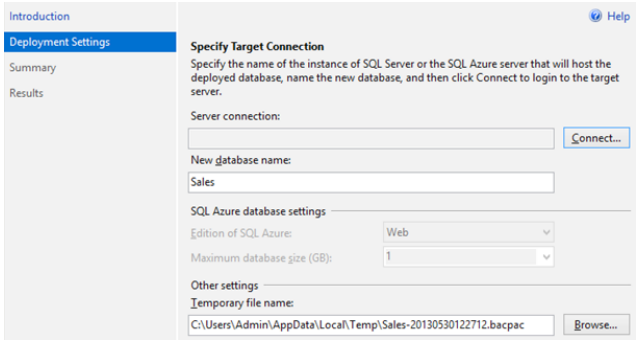


Insert some sample data in each of the two tables so that we can use it for further operations e.g. firing select query when it is exported to SQL Azure.

Step 2: Open SQL Server 2012 Management Studio > right-click on 'Sales' database from Tasks > select 'Deploy Database to SQL Azure' as shown next:



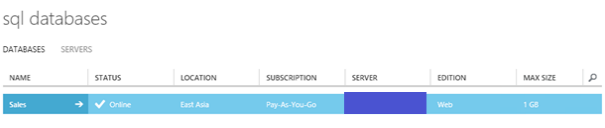
This action will start a wizard for the deployment process. When it shows the 'Introduction' window, click on Next. This will show a 'Deployment Settings' window where the server information needs to be specified for SQL Azure as shown here:



Click on the 'Connect' button to connect to SQL Azure. Use your Azure SQL Server connection string and credentials.

Once connected, the 'Deployment Settings' window will populate the SQL Azure database settings like edition and the database size. It also shows the path of bacpac file. Click on Next, and the Summary window will be displayed where we click on 'Finish' to kick off the export process. Now the Progress window will show the current operation being performed and on successful completion, the result window will be shown with the operations completed during the Database Export. The duration will depend on your internet speed and amount of data in the database.

Step 3: Now switch over to the Azure portal using your subscription. From the management portal, go to Databases where you will find the 'Sales' database created as below:



You can now manage your database on SQL Azure. When you click on 'Manage' on bottom of the page, a Silverlight powered Portal will start, where you need to provide the

login credentials of the database server.

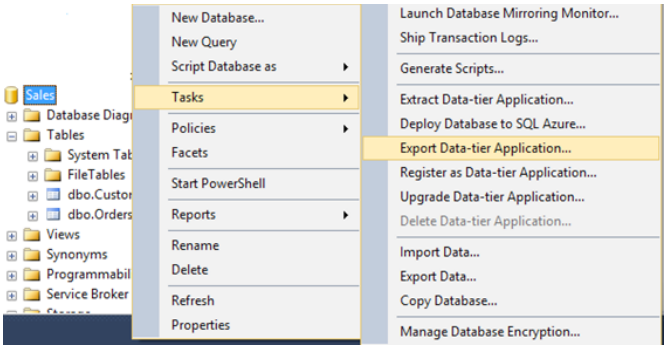
So that's it. The steps that we just saw are the easiest one. Using SQL Server 2012, you can easily migrate your database to the cloud.

Task 2: Performing the Migration using DAC Export to BACPAC using SSMS 2012

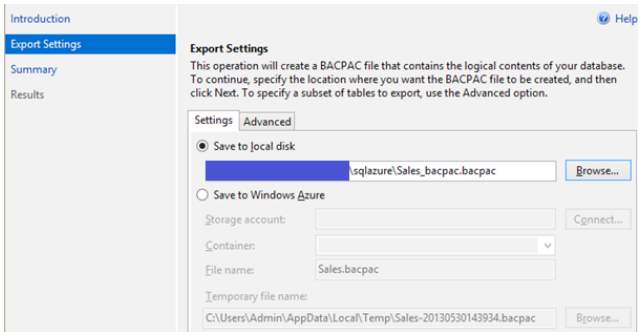
In the previous task, we used the Export Wizard, where once the migration to SQL Azure is started, it cannot be stopped. Now we'll see how we can break this process up into two steps where we first export our on-premises data to a BACPAC file locally and then manually Import the BACPAC file into Azure via the Azure Management Portal.

Step 1 - Using DAC Export

Here we will be using the same 'Sales' database which we have used in the previous Task. Right click on the Sales database and select Task > Export Data-tier Application as shown below:



This will start the 'Export Data-tier' wizard with the Introduction step. Click Next. The 'Export Settings' step will be shown where we have two options - the BACPAC file can be saved on the Local Disk or can also be saved on Windows Azure. This file will now be managed with the help of a Storage account you have on the cloud against your subscription. We will select 'Save to local disk' option, by clicking on Browse button and finding the location where you want to save the file:

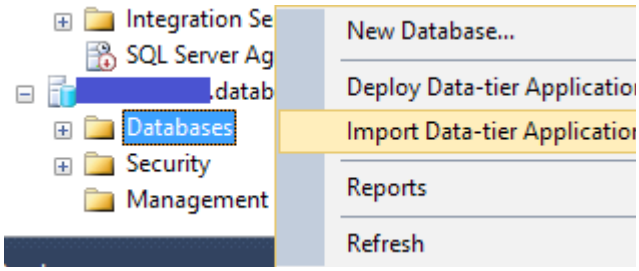


Switch to the 'Advanced' tab and select the Tables you want to Export. Click on Next at the bottom of the wizard page where you will get to the Summary Window. Click on Finish and the Operation Complete window will be displayed. So now you have the .bacpac file locally that has the Database schema and its data.

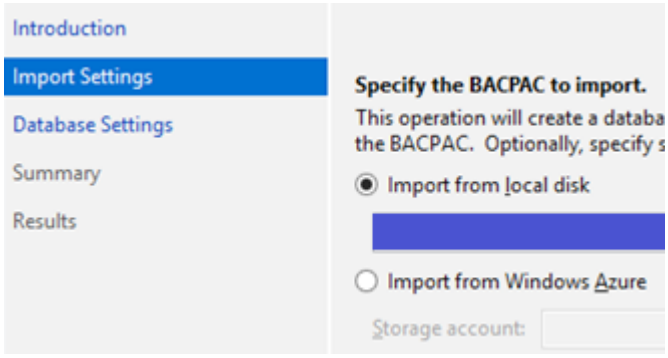
The exported .bacpac file can be imported using two ways.

Step 2 - Import Option A: Open SQL Server 2012 Management Studio (SSMS) and create a new Server Explorer connection to the SQL Azure database server.

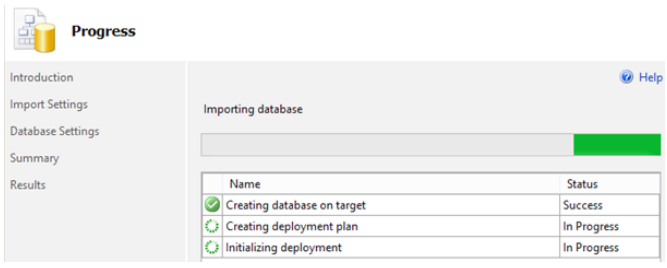
Once the connection is established, right click on the Databases folder and select 'Import Data-tier Application':



This will start the Wizard, click Next on the 'Introduction' window. In the 'Import Settings' window, select the option for 'Import from local disk' as shown below:



Click on next and you will see the 'Database Settings' window. Here you can view the SQL Azure database information like the database server name, database name, edition and size. Now click on Next, you will get the Import Progress as shown below:



After the Import completes, you will see the Result window. Click on Close. Now you will see the Database with tables created on the SQL Azure.

Step 2 – Option b: Importing BACPAC file into Azure directly

a. To perform the Import using the Windows Azure, we will make use of the 'Import' service. However the 'Import' service cannot upload the backup file from local export directly. Instead, we have to re-do the Step 2a above and instead of storing the BACPAC file locally, we'll have to select the option to saving it to Azure.

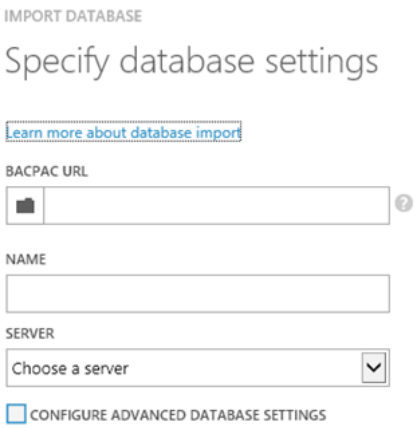
Exporting to Azure will require two things,

- Credentials for an Azure Storage Account
- A Container Name into which the .bacpac file will be uploaded.

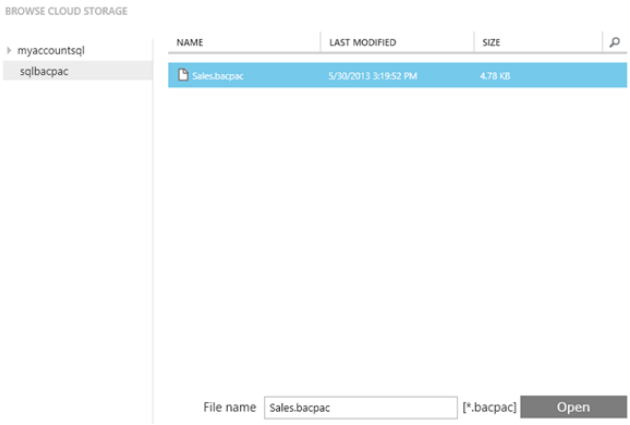
b. Once connected to the Azure Storage account, follow the Wizard till you hit finish and wait for the .bacpac file to be uploaded to Azure Storage.

c. To complete the import, you can now visit the Azure portal and on the database page, click on 'Import' as shown here:

You will see the 'Import Database' window as shown here:

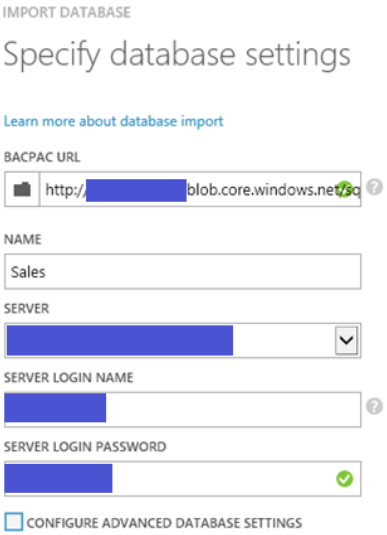


d. Click on the 'Folder' icon on BACPAC URL which brings up the 'Browse Cloud Storage' window. Using it, now you can get the exported file:



e. Once you click on Open, the 'Import Database' window will show the database name. From the Server drop down, you can select the database server name.

f. Thereafter you need to enter the Server Login Name and password as shown here:



g. Once you are done, click on the Check button in the lower right corner of the dialog. The database page will show the database created and you are now good to go again.

So we have seen how to use Import/Export Service to migrate date from a local SQL Server store to a Azure SQL Database.

WORKING WITH SQL REPORTING

Along with SQL Server comes pretty good reporting capabilities in the form of SQL Server Reporting Services

(SSRS). Fact is that Windows Azure SQL database is equipped with similar reporting capabilities. SQL Reporting is a cloud based reporting platform based upon the SSRS. The advantage of using SQL reporting is no additional installations for SSRS is required and hence reduces efforts for maintenance. The key benefits of SQL Azure like availability and scalability are also available with SQL Reporting. To build reports, developer can make use of same tools and these reports are delivered as an integrated part of Windows Azure based solutions.

Comparing SSRS with SQL Reporting

All those who are currently using SSRS will definitely want to know about the similarities and differences between SSRS and SQL Reporting. Some of the differences are:

Criteria	SSRS	SQL Reporting
Tools Used	Business Intelligence Development Studio	Business Intelligence Development Studio
Data Source Supported	Supports Built-in but can also make used of customizable data sources.	Uses SQL Database instance.
Security Modes	Windows Authentication and also supports other authentications.	SQL Database username and password.

Apart from the above differences, there are some rendering based features, plus SSRS reports can be displayed in different formats and these reports can be viewed in browsers. Reports from Report Viewer are provided for WinForms, ASP.NET and SharePoint.

Similarly SQL Reporting based Reports can be viewed in browsers, WinForm and in ASP.NET.

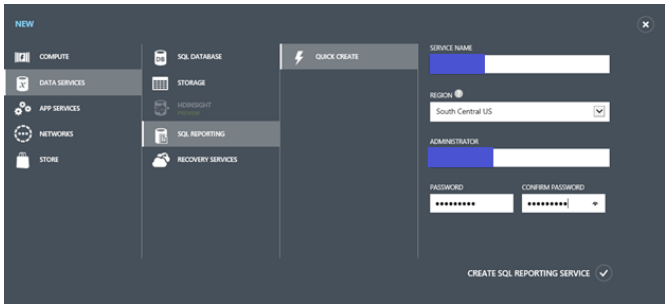
Report management is possible in SSRS using Report Manager in native mode whereas in SQL Reporting we can do this using Windows Azure Management Portal.

Creating SQL Server Reporting Server

First step to get reporting is to setup the Reporting Server. This is relatively easy once logged in to the Azure Management portal.

Step 1: From the portal, click on 'NEW' button in the footer. Pick the following options in the flyout: New Data Service: SQL Reporting: Quick Create.

Now provide the 'Service Name', 'Region', 'Administrator', and 'Password'



Once the service is created, it will be as follows:

The Web Service URL is the location of our Report Server.

Once the report server is up, we can create a Report using Visual Studio 2012.

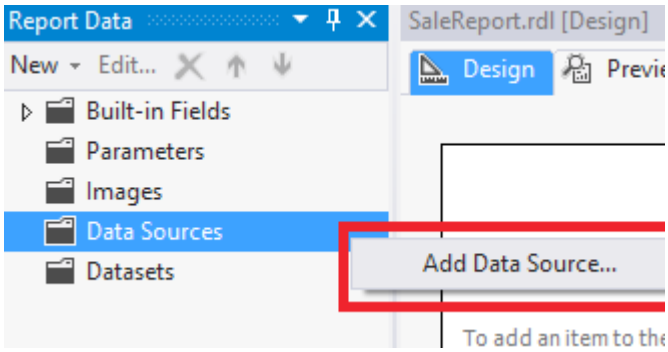
Creating Report

To create report in Visual Studio 2012, you'll need to install Microsoft SQL Server Data Tools - Business Intelligence for Visual Studio 2012. You can download it from Microsoft's download center here.

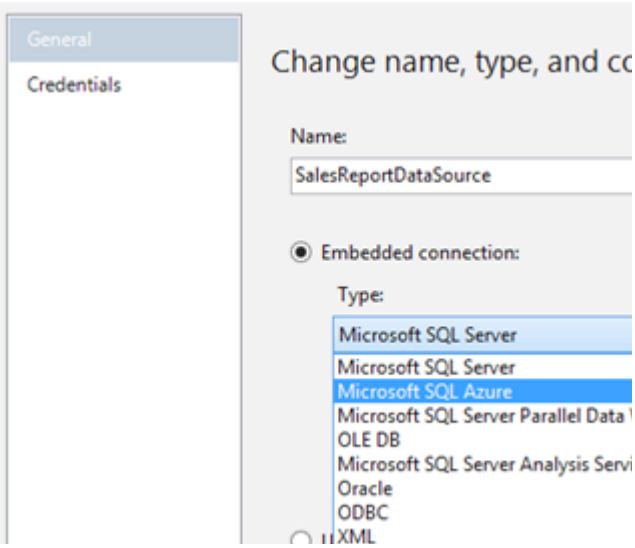
Step 1: Open VS 2012 and create a new Report Server Project using Business Intelligence project templates.

Step 2: Right click on the project and add a new Report in the Project. Once the blank report is added, you will get 'Report Data' tab using which the Data Source can be added.

Step 3: Right-Click on the Data Sources on the 'Report Data' tab and you will get 'Add Data Source' context menu:



Clicking on it brings up the 'Data Source Properties Window'. In this window, add the data source name and in the 'Embedded Connection', select the Type as 'Microsoft SQL Azure'.



Once you select 'Microsoft SQL Azure', you will get the following Connection String Text:

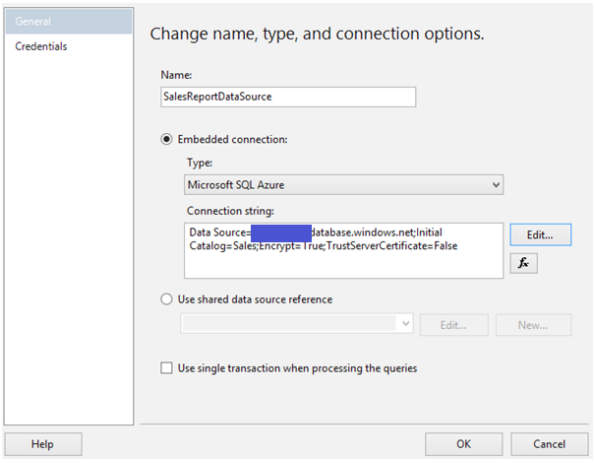
Encrypt=True;TrustServerCertificate=False

These parameters mean the following:

- o Encrypt: Indicates SQL Server will use SSL encryption for all the data sent for communication between client and server.
- o TrustServerCertificate: Indicates that the transport layer will use SSL to encrypt the channel and validate trust by bypassing the walking the certificate chain.

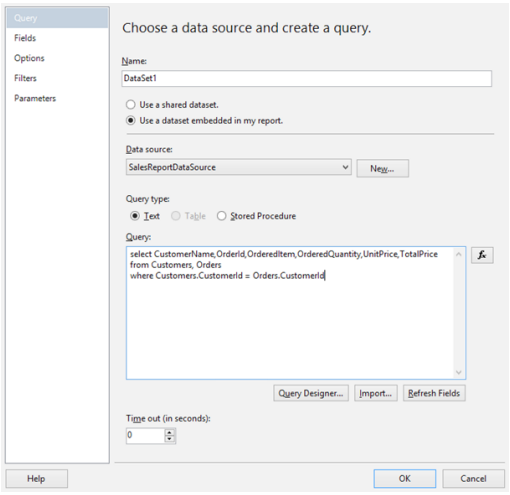
Step 4: Click on the 'Edit' button in the above window, and you will see the 'Connection Properties' window where you need to enter the SQL Azure Database Server name, SQL Authentication Window and select the 'Sales' database from the 'Connect to database' dropdown list.

Once you click on OK, the Data Source Properties Windows will look similar to the following:



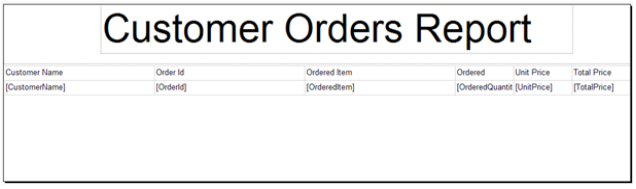
Step 4: The Data Source is now created and we need to create a Data Set. To do that, in the 'Report Data' tab, right click on 'DataSets' node and select 'Add DataSet'. You will see the 'Dataset Properties' window.

Step 5: Select 'Query' from the ListBox from left-side of the window and then select the radio button for 'Use a dataset embedded in my report'. From the dropdown of the 'Data Source', select the data source created in Step 3. In the Query type select the radio button for 'Text', this means that now we need to add T-SQL for fetching data, so add the T-SQL as per your requirement. In the current scenario, we will get Orders Placed by the Customer. After selecting the necessary options, your screen should look similar to the following:

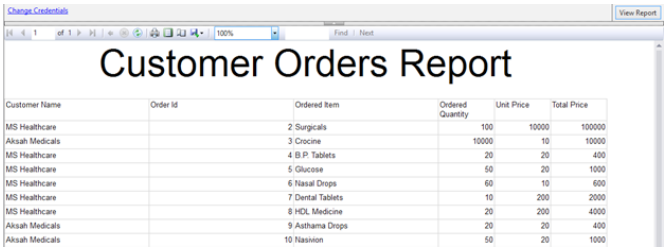


Once you click 'OK', you will be asked to enter credentials for the Data Source where you need to enter SQL Azure Database user name and password. That's it. Now you have created Data Source and the DataSet. Now you can continue with the Report design.

Step 6: Right click on the Report Designer and select Insert-Table. In the table, for each column, drag-drop columns from dataset of name 'DataSet1'. Also add the Header for the Report. After the design, the report will be as shown here:



Step 7: Click on the 'Preview' tab of the report designer, and you will be asked to enter credentials of the DataSource of name 'SalesReportDataSource'. After entering credentials, click on 'View Report' button to display the report.



That's it! This shows us that there is absolutely no difference between SSRS and SQL Reporting as far as Report Designing is concern. Now it's is time for us to deploy the report on the Report server we have created.

Deploying the Report

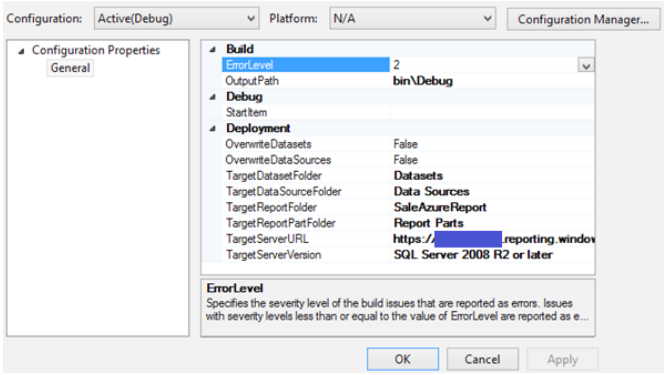
In the section 'Creating Report Server', we have a Web Service URL. We will make use of this URL in the TargetServerURL field for the report project created above.

Step 1: Switch back to the Windows Azure Portal, www.windowsazure.com.

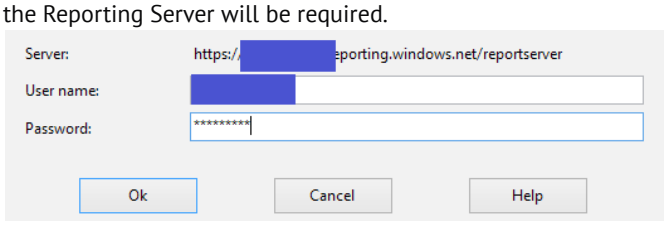
Step 2: Navigate to the 'SQL Reporting' and locate the Web Service URL for the Report server. The URL will be as below:

https://XXXXXXXXX.reporting.windows.net/reportserver

Step 3: In Visual Studio, right-click on the report project created in previous section of 'Creating Report'. In the Property page of the project, set the value of 'TargetServerURL' to the WEB Service URL as shown here:



Step 4: Right-Click on the Report project and select 'Deploy' from the context menu. For the deployment, the credentials for the Reporting Server will be required.



Visual Studio will show the deployment is complete.

Step 5: Once the deployment is completed successfully, now navigate to the URL of the Report Server as shown here: https://XXXXXXXXX.reporting.windows.net/reportserver. You will get the deployed Report link

Click on the 'SaleAzureReport' to deploy the report. You will be asked to enter the credentials. After entering credentials, click on View Report button (top right corner) to display the report : Thus we have created and deployed a Report using the Report Service in Azure.

For SQL Reporting, you can also create Users and define Roles for them. The types of Roles are as following:

- Items Roles:
  - o Browser - Read Access to reports and shared data sources.
  - o Content Manager - Convey permissions for uploading contents like .rdl and .rds. Provides the highest level of permissions on the service, includes the ability of deleting content and add or change permissions and user accounts.
  - o My Reports, Publisher, Report Builder
- System Roles
  - o System Admin and System User.

Once the report is created, you can use it in ASP.NET and in WinForm applications.

Conclusion

Revenue critical enterprise applications often need to be able to scale and handle sudden spurts of data. Building and maintaining such on-premise infrastructure is pretty costly. Cloud services like Azure provide us with flexibility of scalable infrastructure. Today we saw how we could potentially move on-premise databases to Azure and build reports off them. We used the Azure SQL Database and Azure Reporting services for the same ■

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 http://bit.ly/HsS2on

> Biggest challenge I foresee is that Android devices do not have the capability to use TFS API's directly, as they use Java!

Android Client for Team Foundation Server 2012

ALM Guru Subodh Sohoni explores a unique scenario for TFS data access and presents an innovative use case for the newly launched TFS OData Services

It's not common to talk about TFS and Android in the same breath. Both are immensely successful products but without any apparent connection between them. The TFS platform has become ubiquitous for managing software development projects in an Enterprise. At the same time, we have to accept the fact that Android rules the smartphone market and is nudging forward in the tablet world. Given this scenario, we are likely to face the situation where managers and developers may want to use their Android devices to get the status of various artefacts in TFS as well as to trigger builds in TFS. We assume that developers may not use an android device to write code, but they may want to check whether a new work item like a task has been assigned to them or to fire a build and then check it's status. Managers may want to have a look at the entire dashboard of the project, letting them know the status of work items, builds and even the latest check-ins that may have happened.

It is possible to view browser based TFS Web Access application, if the device in use are tablets, with sufficiently large screen.. But it may not be always so and they may use android phone or phablets also to interact with TFS. On

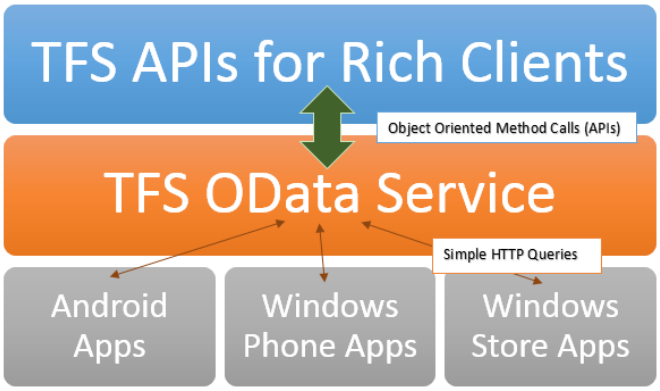
these devices, the web access may not get properly rendered. For these devices, a dedicated application that accesses and shows the TFS data in appropriate format may be a good idea.

CHALLENGES FACED WHILE ACCESSING TFS 2012 FROM ANDROID DEVICES

The biggest challenge that I foresee is that these android devices do not have capability to use TFS APIs directly. There are two reasons for that. The first is that they use Java for application development. The second is that they do not have large memory and processor capability to handle the full-fledged TFS APIs.

The first challenge was addressed by Microsoft by publishing a Java SDK for TFS. It is downloadable from http://www.microsoft.com/en-in/download/details.aspx?id=22616. The second challenge is more difficult to address. Since we may not be able to use the rich set of TFS APIs, we have to look for something different which is light weight on the client side but still leverages the considerable width of TFS data. That *something* is now available as **TFS OData Services**.

TFS OData Services is a platform independent set of REST Services published by Microsoft. It serves as a layer between the TFS and clients that cannot use full set of TFS APIs. Clients can make simple HTTP based queries to the service and TFS OData Service does the translation of that into TFS APIs calls and vice-versa. This reduces the load on the client without affecting TFS at all. It also means that client may use any technology, as long as it can make a HTTP query, it can communicate with TFS, although indirectly.



TFS OData Services can be used in two different environments. First one, is if you are either using TFS Service published by Microsoft or if your TFS is available on the Internet. Second is if you want to access TFS that is purely on premise without any interface to Internet.

PREREQUISITES

The Authentication Story – Hosted TFS

If you want to use TFS OData Service for the hosted TFS solution by Microsoft, it is available from https://tfsodata.visualstudio.com/DefaultCollection. You need to only ensure that client can pass in requisite credentials.

To create such credentials that work with basic authentication, you should create an alias to your account that will allow you to use Basic Authentication. Process to create such alias is as follows:

- Navigate to the account that you want to use on https://tfs.visualstudio.com. For example, you may have https://account.visualstudio.com.
- In the top-right corner, click on your account name and then select My Profile
- Select the Credentials tab
- Click the 'Enable alternate credentials and set password' link
- Enter a password. It is suggested that you choose a unique password here (not associated with any other accounts)
- Click Save Changes

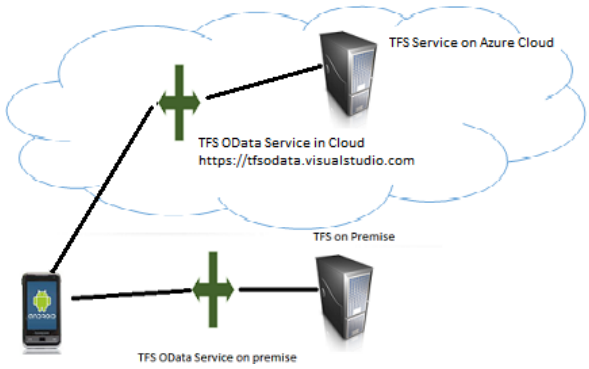
To authenticate against the OData service, you need to send your basic auth credentials in the following domain\username and password format:

- account\username
- password
- Note: **account** is from account.visualstudio.com, **username** is from the Credentials tab under My Profile, and **password** is the password that you just created.

Exposing the OData Service for on-premise TFS

If you want the TFS OData Service to access TFS that is available only on-premise, then you will need to host the TFS OData Service also on-premise. The source code for TFS OData Service is available for download. You can download it from http://www.microsoft.com/en-us/download/details.aspx?id=36230. After downloading that project, you will need to compile the project and then you can host it in your own IIS server within your corporate network. This service allows us to

access any TFS, whether it may be in the Internet or in the private network.



Client side libraries and dependencies

For the client side development, you will need the Android SDK that contains Android Development Toolkit. It is downloadable from here http://developer.android.com/sdk/index.html. It contains the Eclipse version that has the necessary plugin installed to do Android development. It also has a device emulator that can emulate any android device with proper specification.

You will also need a package called Odata4j which is a toolkit published by Google that helps Java applications call and communicate with OData Services easily. This toolkit is downloadable over here.

BUILDING YOUR APP

To begin with, you should start the Eclipse that is bundled with Android Development Toolkit (ADT) that you have downloaded and expanded on your machine. Start an android project. In this project, add the Odata4j JARs as external libraries. Now we are ready to add code to the activity that is provided when the project was created. Name of that activity is MainActivity.

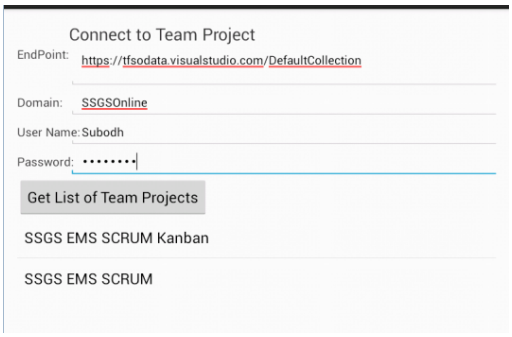
Fetching Data from TFS

In the first activity, we will collect all the data to connect to the TFS OData Service. For my example, I am going to connect to the TFS OData Service that is published by Microsoft at the URL https://tfsodata.visualstudio.com/DefaultCollection. It can accept the Domain that is the account name that I have in TFS Service. In my case it is SSGSOnline. The user name and password that match the basic authentication alias that I have created are also accepted on this activity.

We are going to connect to TFS and show a list of team projects that I have under my account in the TFS Service. In this activity, I have added a ListView to show the list of projects. We define a nested class that extends AsyncTask class. This is a mechanism that is used in Android programming to use multithreading. This class contains a method named doInBackground that gets called when a calling class creates instance of this class and calls the execute method. In the method doInBackground, we can add the code for OData4j objects to access TFS OData Service and indirectly TFS, with my credentials. When the list of projects is available from TFS, we hold it in an ArrayList of strings (called Projects) that is created as a class level variable. Code for doing it is as follows:

```
@Override
protected ArrayList<String> doInBackground(Void...
params) {
//Uncomment following line if you want to debug
//android.os.Debug.waitForDebugger();
try {
 //endpoint is the URL published by TFS OData Service
 // namely https://tfsodata.visualstudio.com/
  DefaultCollection
 Builder builder = ODataConsumers.newBuilder(endPoint);
 builder.setClientBehaviors(new
 BasicAuthenticationBehavior(userName,passwd));
 builder.setFormatType(FormatType.JSON);
 //this is necessary
 ODataConsumer consumer = builder.build();
 List<OEntity> listEntities =
 consumer.getEntities("Projects").execute().toList();
 if (listEntities.size() > 0) {
  for (OEntity entity : listEntities) {
     Projects.add(entity.getProperty("Name").
      getValue().toString());
    }
   }
  } catch (Exception e) { }
 return Projects;
}
```

We show these projects in a ListView.



When a user selects a project, we will call the next activity where we will show various iterations under the selected project.

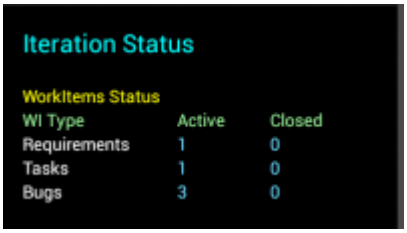
```
@Override
protected void onPostExecute(ArrayList<String> result) {
 super.onPostExecute(result);
 adapter = new
  ArrayAdapter<String>(
    MainActivity.this,android.R.layout.simple_list_
     item_1,
    result);
 list.setAdapter(adapter);
 //this will show the list of projects
 //following code is an event handler for when a
 // project is selected
 list.setOnItemClickListener(new OnItemClickListener() {
 @Override
 public void onItemClick(AdapterView<?> parent,
 View view, int position,long id) {
  String item = ((TextView)view).getText().toString();
  Intent intent = new Intent(MainActivity.this,
  DisplayIterationsActivity.class);
   Bundle bundle = new Bundle();
    bundle.putString("endPoint",endPoint);
  bundle.putString("user", userName);
  bundle.putString("pass", passwd);
  bundle.putString("Project", item);
  intent.putExtras(bundle);
  startActivity(intent);
   }
 });
}
```

Filtering Data using OData Services

TFS OData Service allows us to pass parameters like filter expressions, count, orderby, top/skip, select, format and callback. Syntax for filtering a query is explained in the MSDN article http://msdn.microsoft.com/en-us/library/hh169248(v=nav.70).aspx. Let us use one of them in the Android app:

```
 //get a value of iteration that is bundled from
   earlier activity
 iteration = intent.getStringExtra("iteration");
//TFS OData Service returns iteration path with '<'
   char as separator
 //but requires '\' char as separator in filters in
   queries
 iteration = iteration.replace("<", "\\");
 String StrFilter = "IterationPath eq '" + iteration +
"'";
 // like eq we can also use lt, gt, ne (not equal) etc
in the filter.
 List<OEntity> listEntities =
 consumer.getEntities("WorkItems").filter(StrFilter).
execute().toList();
```

This returns work items in the specified iteration only. We can do computation to get counts and show the data.



For list of various queries supported by TFS OData Service, you can refer to page https://tfsodata.visualstudio.com/Default.aspx It also provides a few queries that can trigger activities on TFS like queuing a build in addition to getting the data.
We just scraped the surface of possibilities using TFS' OData Services. We can continue to leverage all the exposed OData Services and build a compelling dashboard that gives end users a quick view of their projects on TFS.

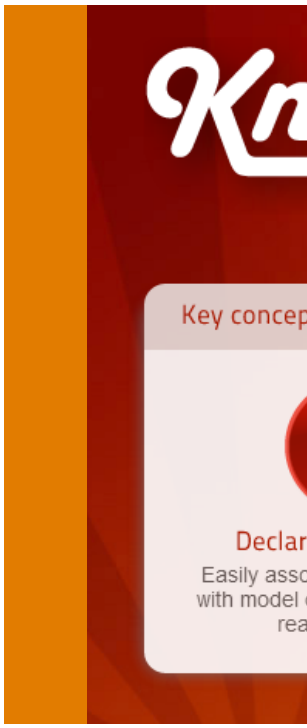
SUMMARY

Although TFS APIs are meant to be used by rich applications on MS Windows Desktop platform, with the advances in mobile technology and use of devices, it has become necessary to access TFS from Android applications. TFS OData Services creates such an opportunity. They provide light weight access to TFS over http/https combined with a powerful mechanism for querying. This should be sufficient to create Android apps that access relevant TFS data and present it innovatively ■

Download the entire source code from our GitHub repository at bit.ly/dncm7-tfsad

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

BUILDING A SORTABLE, PAGED DATATABLE USING KNOCKOUT JS & ASP.NET WEB API



ASP.NET Architecture ***MVP Suprotim Agarwal*** dives into templating and Data Binding using Knockout JS, and applies it to a common Line of Business application requirement – Paged data view in ASP.NET Web API

Knockout JS as we know is a highly versatile JavaScript library that helps us implement the MVVM pattern on the client by providing us with two way data binding and templating capabilities.

Today we will use KnockoutJS and ASP.NET Web API to build an application that represents data in a tabular format with AJAX based sorting and paging capabilities.

Representing Tabular data in a table is as old as web itself and today end users expect a certain amount of features and functionality built in. These include, fast access, sorting, paging and filtering. There are well established libraries like Datatables.net that already do these for us. However, knowing how something is implemented, helps us tweak things to our advantage because no two requirements are the same, ever.

Why KnockoutJS?
Knockout JS simplifies dynamic JavaScript UIs by applying the Model-View-View Model (MVVM) pattern

THE REQUIREMENT

We have a table of user accounts with a few hundred user records. The requirement is to be able to list this user data in a grid layout and provide sorting and paging capabilities on the client side.

Prerequisites

We assume familiarity with ASP.NET Web API and concepts of MVVM. We'll walk through the specifics of Knockout model binding.

GETTING STARTED

We start off in Visual Studio with a new MVC4 project using the Empty template. We will start from scratch and not assume any dependencies.

Installing Dependencies

We start by installing jQuery and Knockout JS using the Nuget Package Management console.


```
PM> install-package jQuery
PM> install-package knockoutjs
```

Next we update our Web API dependencies

```
PM> update-package Microsoft.AspNet.
WebApi
```

We install Entity Framework for the Data Layer

```
PM> install-package EntityFramework
```

We also get BootStrap styles and scripts from Nuget using

```
PM> install-package Twitter.Bootstrap
```

Setting up the Data Layer

Step 1: In the Model folder, add a class called Contact with the following properties.

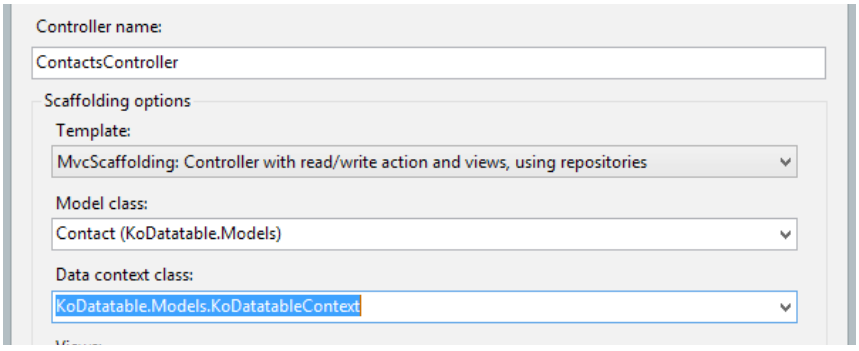
```
public class Contact
{
 public int Id { get; set; }
 public string Name { get; set; }
 public string Email { get; set; }
 public DateTime DateOfBirth { get; set; }
 public string PhoneNumber { get; set; }
 public string City { get; set; }
 public string State { get; set; }
 public string Country { get; set; }
}
```

Step 2: Build the solution. Set up a connection string to point to a LocalDb instance as follows

```
<add  name="DefaultConnection"
 connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename='
|DataDirectory|\KODatatableContext.mdf';Integrated Security=True"
 providerName="System.Data.SqlClient" />
```

You can point to a local SQL Express or SQL Server too, simply update the connection string appropriately.

Step 3: Next we scaffold a Controller, Repository and Views using MVC Scaffolding



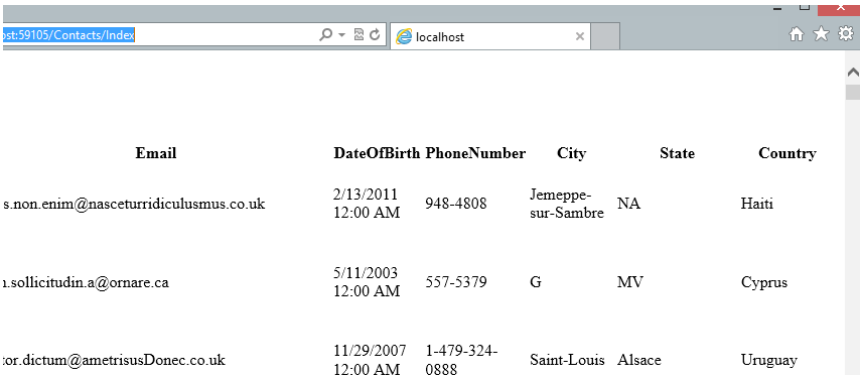
Step 4: In the generated KoDatatableContext class, add the following constructor.

```
public KoDatatableContext():base("DefaultConnection")
{
}
```

This forces the DB Context to use the connection string. If you don't provide this, EF looks for a SQL Server Express at .\SQLEXPRESS and creates a DB Based on the Context's namespace.

Step 5: Run the application and visit the /Contacts/Index page. This will first generate the DB and then bring up the empty Index page.

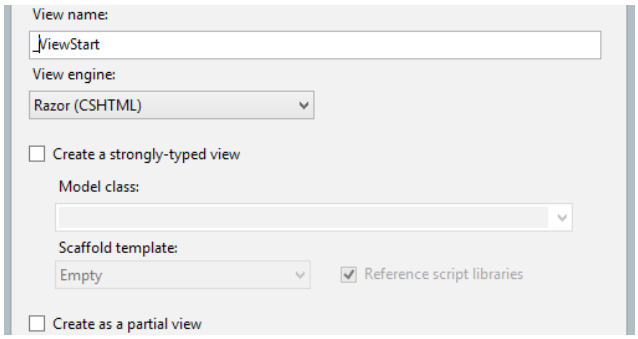
Step 6: To have multiple pages of contacts, we need some pre-generated data. I found http://www.generatedata.com to be a pretty neat site for this purpose. I added all the columns except Id and requested it to generate 100 rows into an Insert Script. Once the script is downloaded, connect to the DB using the Database Explorer in Visual Studio and run it on the KnockoutDataTableContext DB. Refresh the /Contacts/Index page and you should see 100 rows of data.



Sprucing it up with BootStrap

The Index page looks rather plain vanilla so let's get BootStrap kicking and spruce it up a little.

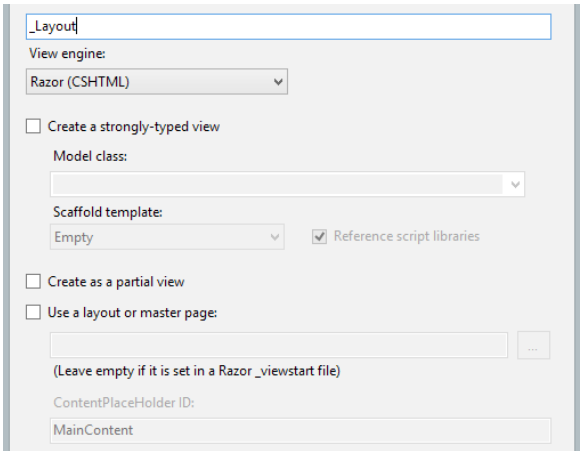
Step 1: Add _ViewStart.cshtml in the Views folder.



Update the markup to contain only the following

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Step 2: Create a folder /Views/Shared and add _Layout.cshtml to it.



Step 3: In the App_Start folder add a BundleConfig.cs class. To make use of Bundling and Minification, we need to add one more Nuget package

```
PM> install-package Microsoft.
AspNet.Web.Optimization
```

Once the optimization bundle is downloaded, setup BundleConfig.cs. In App_Start folder add a BundleConfig.cs

```
public class BundleConfig
{
 public static void
RegisterBundles(
BundleCollection bundles)
 {
  bundles.Add(new ScriptBundle("~/bundles/jquery").
  Include(
   "~/Scripts/jquery-{version}.js"));
  bundles.Add(new ScriptBundle(
  "~/bundles/bootstrap").Include(
   "~/Scripts/bootstrap.js",
   "~/Scripts/html5shiv.js"));
  bundles.Add(new ScriptBundle("~/bundles/jqueryui").
  Include(
   "~/Scripts/jquery-ui-{version}.js"));

  bundles.Add(new ScriptBundle(
  "~/bundles/jqueryval").Include(
```

```
    "~/Scripts/jquery.unobtrusive*",
    "~/Scripts/jquery.validate*"));
   bundles.Add(new ScriptBundle("~/bundles/knockout").
   Include("~/Scripts/knockout-{version}.js"));
   bundles.Add(new StyleBundle("~/Styles/bootstrap/
   css").Include("~/Content/bootstrap-responsive.css",
    "~/Content/bootstrap.css"));
 }
}
```

This sets up BootStrap and jQuery script and style bundles.

Step 4: Update the _Layout.cshtml to use the bootstrap CSS and Script bundles

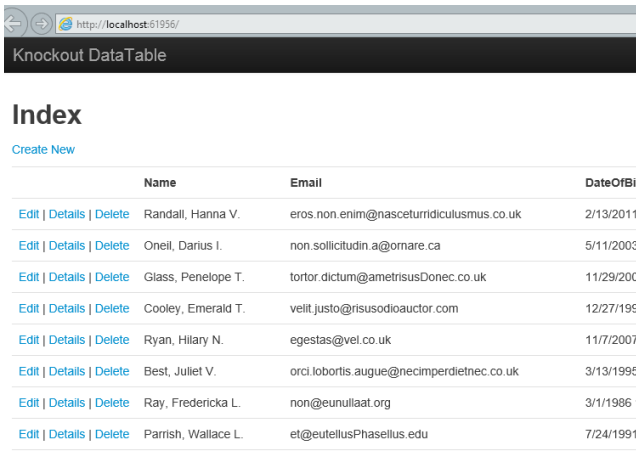
```
@{
    ViewBag.Title = "Knockout DataTable";
}

<!DOCTYPE html>
<html>
<head>
 <meta name="viewport" content="width=device-width"
/>
 <title>@ViewBag.Title</title>
 @Styles.Render("~/Styles/bootstrap/css")
</head>
<body>
 <div class="navbar navbar-inverse">
  <div class="navbar-inner">
   <div class="brand">
    @ViewBag.Title
   </div>
  </div>
 </div>
 <div class="container-fluid">
  <div class="row-fluid">
   @RenderBody()
  </div>
   @Scripts.Render("~/bundles/jquery")
   @Scripts.Render("~/bundles/knockout")
   @Scripts.Render("~/bundles/bootstrap")
   @RenderSection("Scripts", false)
  </div>
 </body>
</html>
```

Step 5: In the Index.cshtml add the class table to the container

```
<table class="table">
…
</table>
```

If we run the app now, it should look much cleaner



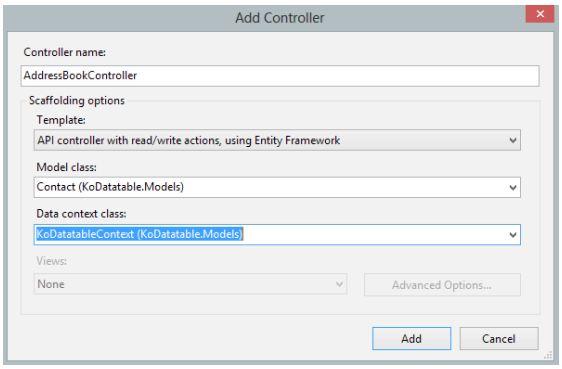
We are all set, let's implement the rest of the features.

PAGING

Switching to Web API

Since we want all of the rendering to be done on the client side, instead of using the MVC Controller and rendering the View on the server, we'll switch to Web API and change the markup to use Knockout JS.

Step 1: Add a new API Controller called AddressBookController



Step 2: Update the existing Contacts controller so that it doesn't return any data

```
public ViewResult Index()
{
    return View();
}
```

Adding KO ViewModel and Updating UI

We will step through the KO Script incrementally. To start off with, we add a new JavaScript file called ko-datatable.js under the Scripts folder. This will contain our view model and data-binding code.

Step 1: As a first step, we setup our viewModel to contain only the contacts collection as an observable collection. When the document is loaded, it calls our AddressBook api and gets the list of contacts. It then pushes them into the contacts viewModel property. Finally we call ko.applyBindings(...) to do the data binding.

```
/// <reference path="knockout-2.2.1.js" />
/// <reference path="jquery-2.0.2.js" />
var viewModel = function ()
{
 $this = this;
 $this.contacts = ko.observableArray();
}

$(document).ready(function ()
{
 $.ajax({
  url: "/api/AddressBook",
  type: "GET"
 }).done(function (data)
 {
  var vm = new viewModel();
  vm.contacts(data);
  ko.applyBindings(vm);
 });
});
```

Step 2: To utilize the ViewModel data, we update the Index.cshtml to use KO Databinding syntax instead of the previous server side razor implementation.

We use the foreach data-binding to bind the contacts collection to the table (body), whereas we bind each td to the field names.

```
<tbody data-bind="foreach: contacts">
 <tr>
  <td data-bind="text: Name"></td>
  <td data-bind="text: Email"></td>
  <td data-bind="text: DateOfBirth"></td>
  <td data-bind="text: PhoneNumber"></td>
  <td data-bind="text: City"></td>
  <td data-bind="text: State"></td>
  <td data-bind="text: Country"></td>
 </tr>
</tbody>
```

At this point if we run the app, we'll see the same type of table layout we had seen for the server side binding. The only difference now is it's using Knockout's Databinding and templating to generate the table on the client side.

Implementing Pagination

For paged data, we need to be able to track the following:

- Current Page Index: Starts at zero and increases/decreases as user selects different pages
- Page Size: A value indicating the number of items per page
- All elements: The complete list of elements from which the current page will be sliced out.

We add the above properties in our view Model and add a KO Computed observable to calculate the current page. Computed Observables in KO are functions that are recalculated automatically when the observables used in the function change. So our computed observable returns a slice of the array elements from the complete list of Contacts. By default page size is 10 and current page index is 0, so the first 10 records are shown.

There are two functions previousPage and nextPage. These are bound to the Previous and Next buttons' click events. The next button circles back to the first page once it reaches the last page. Similarly the previous button circles back to the last page, once it is reaches the first page.

```
var viewModel = function ()
{
 $this = this;
 $this.currentPage = ko.observable();
 $this.pageSize = ko.observable(10);
 $this.currentPageIndex = ko.observable(0);
 $this.contacts = ko.observableArray();
 $this.currentPage = ko.computed(function ()
 {
  var pagesize = parseInt($this.pageSize(), 10),
  startIndex = pagesize * $this.currentPageIndex(),
  endIndex = startIndex + pagesize;
  return $this.contacts.slice(startIndex, endIndex);
 });
```

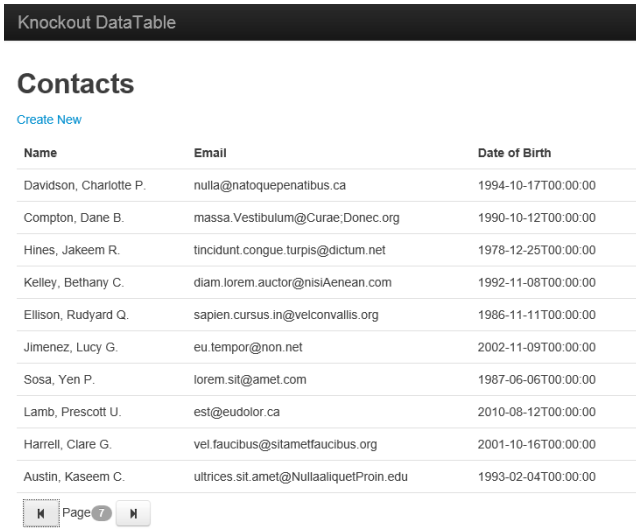
```
 $this.nextPage = function ()
 {
  if ((($this.currentPageIndex() + 1) *
  $this.pageSize()) < $this.contacts().length)
  {
   $this.currentPageIndex(
           $this.currentPageIndex() + 1);
  }
  else
  {
   $this.currentPageIndex(0);
  }
 }

 $this.previousPage = function ()
 {
  if ($this.currentPageIndex() > 0)
  {
   $this.currentPageIndex(
           $this.currentPageIndex() - 1);
  }
  else
  {
   $this.currentPageIndex((Math.ceil(
   $this.contacts().length / $this.pageSize())) - 1);
  }
 }
}
```

We add the following Table footer markup for the previous, next and current page index.

```
<tfoot>
 <tr>
  <td colspan="7">
   <button data-bind="click: previousPage"
     class="btn">
   <i class="icon-step-backward"></i></button>
    Page<label data-bind="text: currentPageIndex()
     + 1" class="badge"></label>
   <button data-bind="click: nextPage" class="btn">
   <i class="icon-step-forward"></i></button>
  </td>
 </tr>
</tfoot>
```

If we run the app now, the previous & next buttons will be functional and we can see the data change accordingly.



SORTING

We can implement sorting on the client side as well with the help of Knockout's sort function for the observable array. KO provides a sort function with two inputs left and right that contain the JavaScript objects from the array. We can write custom logic to sort based on any of the properties in the object. So if we wanted to sort on the Name property in our Contact object, our sorting function would be as follows:

```
$this.contacts.sort(function (left, right)
{
    return left.Name < right.Name ? 1 : -1;
});
```

However, the above is only a one-way sort, as in after it's sorted, say in 'ascending' order, there is no toggling to 'descending' order. All sortable data table implementations toggle the sort order on clicking the header. To be able to toggle between ascending and descending, we have to store the current status in our ViewModel.

Another issue is that the Name property is hard coded above. We would like to sort based on the column header we click on. So instead of the function picking the Name property, we should determine the Property to be sorted on at run time.

UPDATING THE MARKUP

We update our Index.cshtml's table header as follows

```
<thead>
 <tr data-bind="click: sortTable">
  <th data-column="Name">Name
```

```
  </th>
  <th data-column="Email">Email
  </th>
  <th data-column="DateOfBirth">Date of Birth
  </th>
  <th data-column="PhoneNumber">Phone Number
  </th>
  <th data-column="City">City
  </th>
  <th data-column="State">State
  </th>
  <th data-column="Country">Country
  </th>
 </tr>
</thead>
```

We have bound the click event of the table's header row to a function called sortTable in the ViewModel. We will see the implementation of this function shortly.

Next we have used HTML5 data- attributes to define a new attribute called data-column and set the value to the name of each property that column is bound to.

Updating our ViewModel and Implementing the Sort

1. First we add a property that saves the current Sort type (ascending or descending) and set it to 'ascending' by default.

```
$this.sortType = "ascending";
```

2. Next we add the function sortTable that does the actual sorting

```
$this.sortTable = function (viewModel, e)
{
 var orderProp = $(e.target).attr("data-column")
 $this.contacts.sort(function (left, right)
 {
  leftVal = left[orderProp];
  rightVal = right[orderProp];
  if ($this.sortType == "ascending")
  {
   return leftVal < rightVal ? 1 : -1;
  }
  else
  {
   return leftVal > rightVal ? 1 : -1;
  }
 });
 $this.sortType = ($this.sortType == "ascending") ?
"descending" : "ascending";
 }
};
```

This code works as follows:

a. The function first extracts the column on which the click happened. It uses the event object 'e' and picks out the value in the 'data-column' attribute. As we saw in the markup, 'data-column' has the name of the property to which that column's data is bound to.

b. Once we have which column to sort on, we call the 'contacts' observable array's sort method with a custom delegate that returns 1 or -1 based on the comparison result.

c. In the sort function, we extract the property using the column name. Since this is now coming from the 'data-column' attribute, we will automatically get the correct property name to compare.

d. Next, based on whether the current sortType is ascending or descending, we return the value 1 or -1.

e. Once the entire array has been sorted, we set the sortType attribute to opposite of what it was.

That's all that needs to be done to get sorting going! If you run the application now, you can click on the columns and see sorting in action on each.

Showing a Sort-Indicator arrow

Usually when we have a sortable table, the column on which it is sorted has an arrow indicating if it is sorted in ascending or descending order. In our view model, we are already saving the sortType, we will add two more properties.

- First is iconType - this is set to the bootstrap css style 'icon-chevron-up' or 'icon-chevron-down'. This is set in the sortTable function when the user clicks on the header column.

- Second property is for saving the name of the column that the table is currently sorted on – currentColumn. This value is also set in sortTable function.

```
$this.sortTable = function (viewModel, e)
{
```

```
 var orderProp = $(e.target).attr("data-column")
 $this.currentColumn(orderProp);
 $this.contacts.sort(function (left, right)
 {
  …
 }
 …
 $this.iconType(($this.sortType == "ascending") ?
"icon-chevron-up" : "icon-chevron-down");
}
```

Next we update the Table header markup, to do two things:

1. Set the arrow visible based on the currentColumn and invisible for the rest of the columns.

2. Set the arrow to ascending or descending based on sortType To do this we do the following:

a. Add a couple of CSS styles that we'll set dynamically

```
<style type="text/css">
    .isVisible {
        display: inline;
    }

    .isHidden {
        display: none;
    }
</style>
```

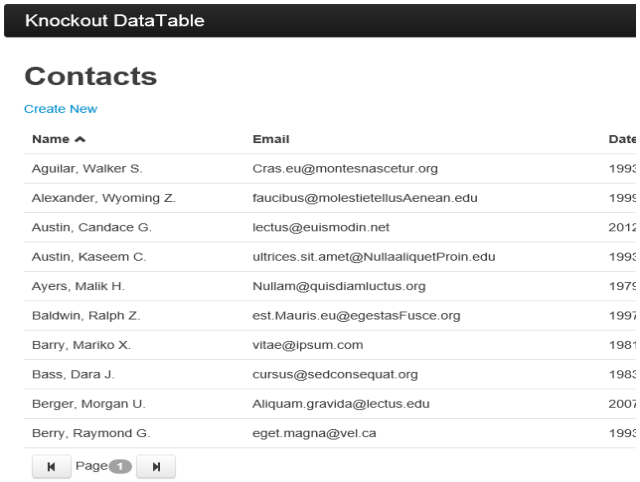
b. In the table header, for every column, we add a span whose visibility is set based on whether the currentColumn value is the same as the data-column attribute. For the Name column, if the currentColumn() value is 'Name' then we set the class to 'isVisible' which in turn makes the span and its contents visible. For all other values of currentColumn(), the span remain hidden.

```
<table class="table">
 <thead>
  <tr data-bind="click: sortTable">
   <th data-column="Name">Name
    <span data-bind="attr: { class: currentColumn()
     == 'Name' ? 'isVisible' : 'isHidden' }">
     <!-- add image here -->
    </span>
   </th>
   …
  <tr>
 </thead>
</table>
```

c. Finally we add an icon in the placeholder above and set the image by binding the class attribute to iconType

```
<i data-bind="attr: { class: iconType }"></i>
```

If we run the application now, we can see the sorting and the sort indicator in action. For example, the snapshot below shows that the table is sorted in ascending order of the Name column.



DYNAMIC PAGE SIZE

The final feature that we'll add is to let users select the number of rows per page. Currently it is hard-coded to 10. We will add an html select to the footer with various other sizes and bind it to the pageSize property of our ViewModel.

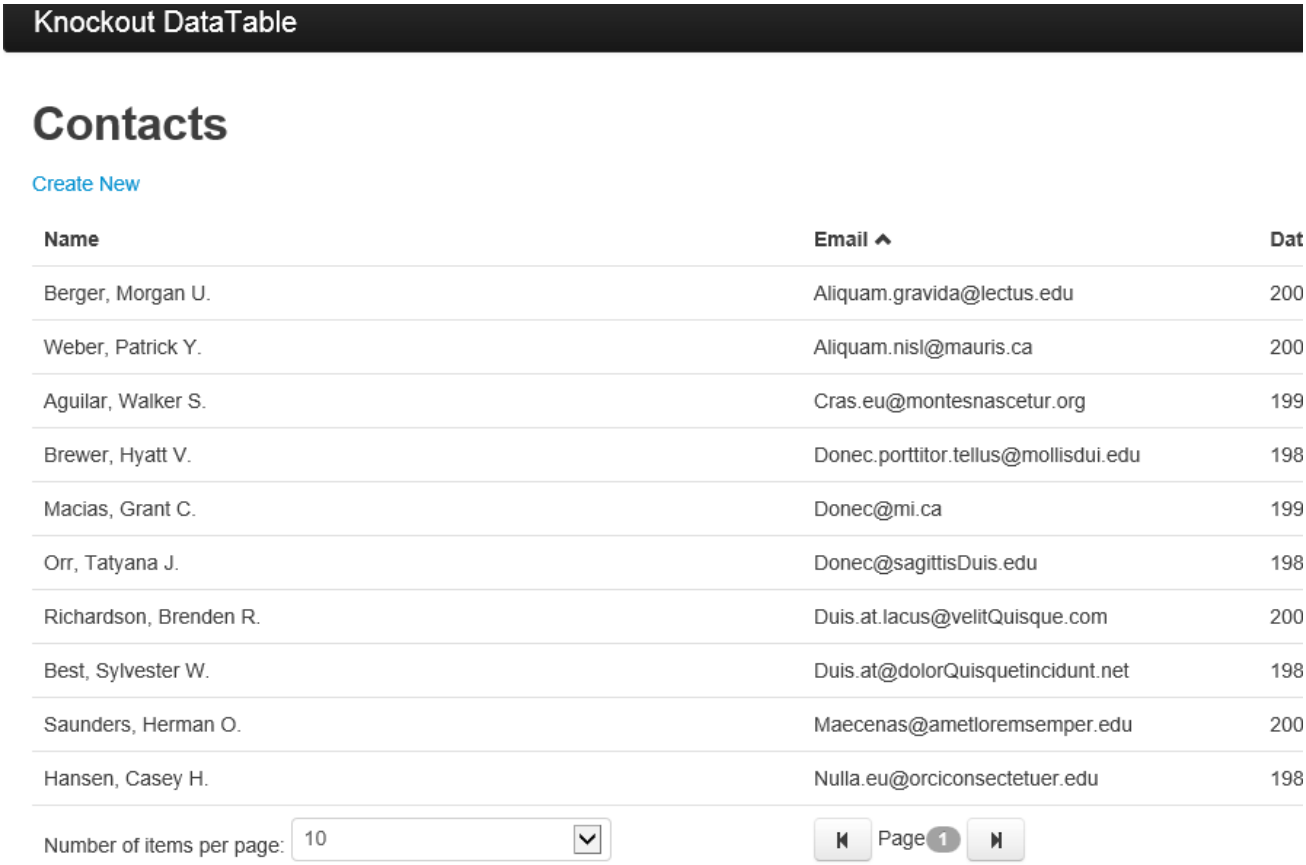
Because the currentPage property is computed and one of the observables it is computed on is pageSize, any change in pageSize will trigger the re-computation of currentPage property resulting in re-calculation and rendering of the table.

As shown in the markup below we have page sizes from 10 to 100 in steps of 10

```
<tfoot>
 <tr>
  <td> Number of items per page:
   <select id="pageSizeSelector" data-bind="value:
pageSize">
    <option value="10">10</option>
    <option value="20">20</option>
    <option value="30">30</option>
    <option value="40">40</option>
```

```
    <option value="50">50</option>
    <option value="60">60</option>
    <option value="70">70</option>
    <option value="80">80</option>
    <option value="90">90</option>
    <option value="100">100</option>
   </select>
  </td>
 …
 </tr>
</tfoot>
```

Here is our final Table



With that have completed our requirements of having a Sortable grid with pagination.

CONCLUSION

We saw how to use Knockout JS for templating and data-binding to get pretty rich functionality with minimal code.

We can certainly improve on this model. For example, if we have more than a few hundred rows of data, we should use server side data selection and instead of bringing back the complete dataset, use only the current page's data. This will offer the right balance between performance and richness of features.

Overall Knockout is a powerful library and can be used as a drop-in enhancement to bring in rich functionality on the client side ■

Download the entire source code from our Github repository at bit.ly/dncm7-kosdt

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