ISSUE 06 | MAY-JUNE 2013 EDITION

DNCMAGAZINE

www.dotnetcurry.com

Headfirst into Web API

Remember your Hot Towel when going to the SPA!

ASP.NET MVC

Responsive Image Viewer using CSS3 Media Queries

Extending Team Foundation Server 2012

Push Notifications in Windows 8 Apps!

14 IT Admin Skills for .NET Developers

INTERVIEW with ERIC LIPPERT

Real-time network applications in .NET using SignalR

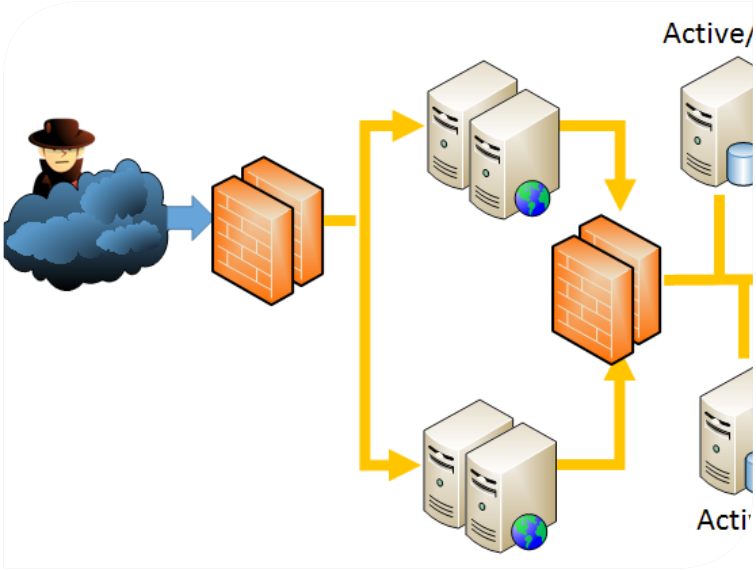
SharePoint Server 2013 Workflow

Document Driven ALM using TFS APIs

MAY-JUNE 2013

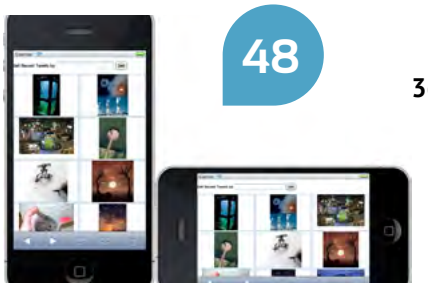
CONTENTS

ENTERPRISE



WEB

CUTTING EDGE



48

INTERVIEW

LETTER FROM THE EDITOR

Hello Technology lovers, welcome to the sixth edition of DNC Magazine. The tech world is alive with volatility, Apple's stocks are suddenly down, Microsoft's stocks are up and Google has turned wearable computing into reality with the limited launch of its ground breaking project 'Glass'.

Back at DNC Magazine, as always, we have new tech-topics, new authors and lots of exclusive content. This month we have *Fanie Reynders* joining our exclusive club of contributors, with his quickfire introduction to the Hot Towel SPA template for ASP.NET MVC. Welcome Fanie!

We talk to C# legend *Eric Lippert* in our interview section and get deep into his journey at Watcom, Microsoft and as a member of the C# language team. Don't miss out on the C# tricks he shares with us at the end of the interview! *Omar Al Zabir* shares some golden IT Administration rules for all who are serious about their production deployments. His insights come from years of hands on experience. This one is a must read! Then we have ALM MVP, *Subodh Sohoni* sharing with us various TFS Customization options.

Author, MVP and C# lover, *Filip Ekberg* does a real deep dive into Real Time systems and shows us how build one in .NET using SignalR. We have *Pravinkumar* showing us what's new and cool with Workflows in SharePoint 2013.

We also have contributions from ASP.NET MVP, *Suprotim Agarwal* on Responsive Design and CSS and Web API Content formatters and one contribution from *yours truly* on Azure Mobile Services and Push notifications.

Hope you enjoy the content and do keep the feedback coming!

Sumit Maitra
Editor in Chief

www.dotnetcurry.com
dnc mag

Editor In Chief • Sumit Maitra
sumitmaitra@a2zknowledgevisuals.com

Editorial Director • Suprotim Agarwal
suprotimagarwal@dotnetcurry.com

Art Director • Minal Agarwal
minalagarwal@a2zknowledgevisuals.com

Contributing Writers • Fanie Reynders, Filip Ekberg, Omar Al Zabir, Pravinkumar Dabade, Subodh Sohoni, Sumit Maitra, Suprotim Agarwal

Advertising Director • Suprotim Agarwal
suprotimagarwal@dotnetcurry.com

Interview Feature • Eric Lippert
Twitter @ericlippert

Image Credits • Rob Gruhl, Bob Archer, Neersighted

Next Edition• 1st July 2013
www.dotnetcurry.com

ESSENTIAL IT ADMIN SKILLS FOR .NET DEVELOPERS

PLANNING BUDGETING AUTOMATION

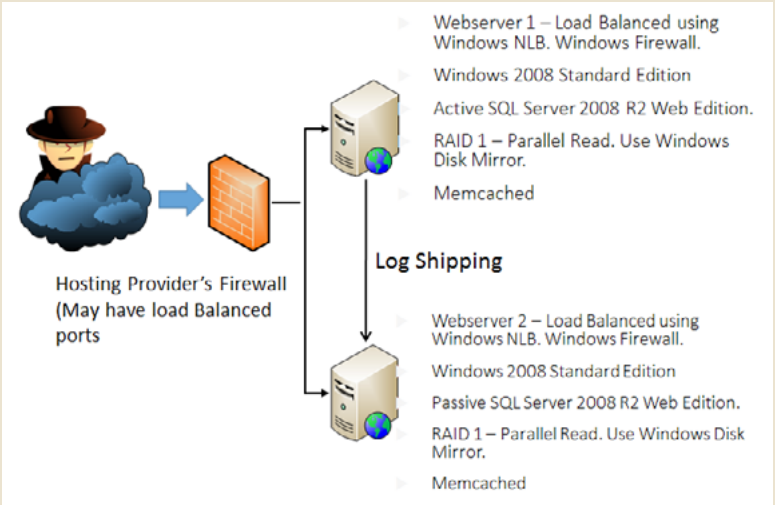
You have spent a great deal of time building an awesome .NET software. Now you need to deploy it on production hardware, but how? *Omar Al Zabir* shares some essential IT Admin skills that every .NET developer should have before going live.

You have built an awesome .NET application. Now you have to deploy it and scale it to millions of users. How do you choose the right production architecture for 99.99% availability? How do you make sure your website is configured for optimum performance? How do you regularly tune the database to keep it in top notch shape without downtime? In this article, I will share some Admin skills that I have learnt while building my startup from hundreds to million of users and how I have overcome the most frequent production challenges, high volume .NET web apps face.

01 DESIGNING A RELIABLE SUPER CHEAP PRODUCTION ARCHITECTURE

Say you are still a garage startup and money is your primary concern. You want to spend the least amount of money to buy hardware, yet you want to have a decent reliability around your setup. So even if one server goes down, your website should be able to recover within minutes.

Let's look at the cheapest possible production architecture money can buy, that offers some decent reliability and survives a complete server failure. See the diagram here:



HIGHLIGHTS FROM THIS ARCHITECTURE:

• Both servers are Windows 2008 R2 web/standard editions to save cost.

• Both servers have SQL Server Web Edition installed provided by hosting company. If you want to buy your own servers, then you have to purchase Standard Edition.

• One server is called Primary Database Server that has the databases on it.

• The other standby server has standby copies of the databases configured through SQL Server's Log Shipping feature.

• Both servers have IIS and web app is running on both.

• Both servers have Windows Network Load Balancing configured to load-balance the web traffic between both servers. NLB is configured on port 80 to distribute web traffic equally on both servers. Alternatively, you can ask your hosting provider to give you two load balanced ports on their firewall or load balancer.

• Both servers have two disks and Windows Disk Mirroring is configured to provide a software RAID1 solution. This way, both disks have the exact same data. Even if one disk fails, there will be no disruption.

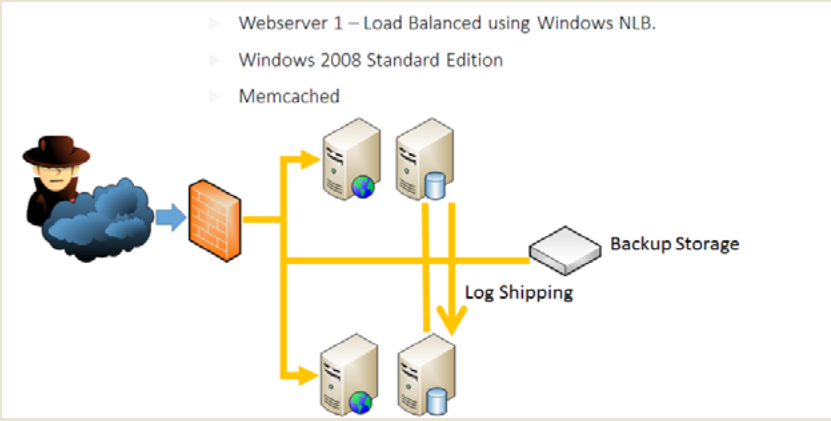
This configuration gives you pretty decent reliability. If one server fails, the web traffic will be automatically diverted to the other server. But the SQL Server Standby databases have to be manually brought online and you need to change the web.config on the surviving server, to point to the surviving SQL Server.

However, you should go with this super cheap configuration only if you are a garage Startup and you are funding from your own pocket. If you want decent reliability with high performance and availability too, you need to go for the next architecture.

02 DECENT PRODUCTION ARCHITECTURE FOR 99% AVAILABILITY

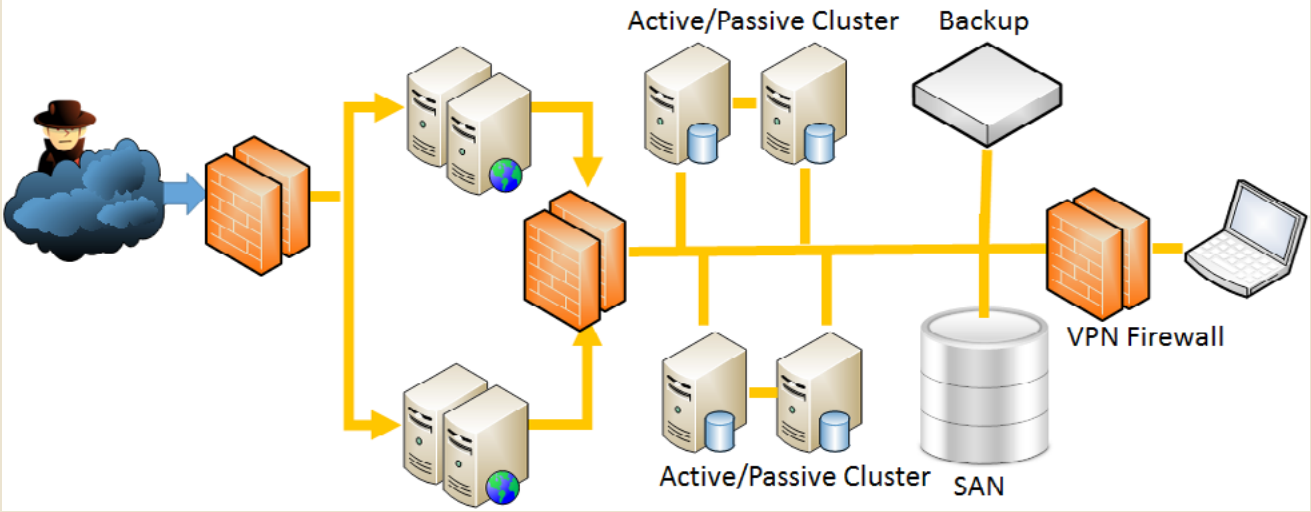
The first thing you need to do is separate out the databases into their own servers. Whenever you put SQL Server on a server, you will see all your RAM gets consumed too quickly. It is designed to allocate all available memory.

So, IIS and SQL Server will be fighting for RAM unless you go to the SQL Server Management Studio and fix the maximum memory limit. But you shouldn't do that unless you are desperate. You should put SQL Server on its own server with as much RAM as possible. The best thing would be to have the same amount of RAM as the size of data in your database. But RAM is expensive!



03 SUPER RELIABLE HOSTING ARCHITECTURE FOR 99.99% AVAILABILITY

This is where things gets really expensive but rock solid. If you have a business critical application that earns millions per year and has to be up and running 24x7, you need an architecture that looks like this:



There is enough redundancy at every level. First, there are redundant firewalls. Yes, that's the last thing you think about – having a standby firewall. But once we had this nightmare where our one and only firewall was down and as a result, the entire website was down. We learnt the hard way to have a passive firewall since then. Then you have enough web server availability so that even if 2 of them are down, you will have no issues serving the traffic.

A common mistake IT guys make while designing production architecture is – they do capacity analysis to find out how many servers they need and they add one server to it, and buy those many servers. But when you have one server down, you might have to release a critical patch on your web app. During that time, you will have to take one server out at a time and patch them one after another. At that moment, you will have to serve traffic while both the servers are down. That's why whenever you calculate how many servers you need to serve the traffic as per the projected volume, always ensure you can serve that traffic even if two servers are down.

The other important addition in this architecture is SAN – Storage Area Network. Hosting providers have their SAN, which is like a monstrous collection of disks controlled by super awesome disk controllers. Servers connect to SAN via fibre channel for lightning fast Disk I/O. You purchase volumes from SAN. For e.g., you ask for 5x200 GB volumes from the SAN. Then you get those volumes available in your server and the server can read- write on those volumes. SAN is very expensive. Only databases and highly critical data are stored on SAN.

SAN offers the maximum reliability and performance. They are faster than local disks. SAN can offer you complete disk fail protection. Moreover, SAN offers on-the-fly increase of disk volumes. If your database is growing faster than you can handle and you are about to run out of disk space, you can increase the size of the volume, *on-the-fly.*

04 CHECKLIST FOR CREATING IIS WEBSITES

Here are a couple of things we always do while creating a website on IIS:

• Create each website in its own pool, using App Pool identity. This creates a new user for each app pool. This way, we can give granular permissions to certain App Pool users and do not have to meddle with the NETWORK_SERVICE account.

• Create the App_Data folder and allow write access to

NETWORK_SERVICE,IUSR and IISAPPPOOL\<ApplicationPoolName>.

• Enable Static and Dynamic Compression.

• Turn on Content Expiration from IIS > Website > HTTP headers > Common Headers. Set to 30 days. This makes all the static file cacheable on the browser. You will get significant reduction in web traffic when you turn this on.

• From IIS > Website > Bindings, we map both www.yourdomain.com and yourdomain.com. Alternatively we setup a redirector website to redirect traffic on yourdomain.com to www.yourdomain.com. The later is better because this way users are never browsing your website over yourdomain.com and they will always be on www.yourdomain.com. If your website is such that users copy and share links to pages frequently, you should go for such redirection.

• From IIS Logging, turn on "Bytes Sent" and "Bytes Received" fields. Change creating log files to hourly ,if you have heavy traffic. This way each log file size will be within the manageable size and if you have to parse the log files, it won't take too long. Whenever we have to diagnose slow pages, we first look into the IIS log and ensure if a page is taking a long time to execute, and it is not because of large "Bytes Sent".

• Map 404 to a well defined error page from IIS > Website > Error Pages > 404. You can set it to redirect to your websites homepage.

• Copy website code to each server and synchronize the files last modification date and time. This way, each file will have the exact same last modified date time on each server. That means, if a browser gets jquery.js file from webserver1, and it tries to hit webserver2 on another page visit and asks webserver2 about the last modified datetime of jquery.js, it will get the exact same date time and it won't download the whole file again.

• Create a static website hosted on a different domain. Eg. Staticyourdomain.com and have all static files served from this domain. It prevents large ASP.NET cookies from being sent over static files.

These are some techniques we have learnt over the years to avoid common mistakes and tune websites to serve well cached traffic to the browser, delivering faster page load performance.

05 REMOVING UNWANTED HTTP HEADERS

There are some HTTP response headers that make hackers lives easier in order to detect what version of IIS you are running on and what .NET version you are on. You can remove most of these headers using web.config. But there are some that cannot be removed using the config file and you need to write a custom HttpModule to remove those from every response; especially the ETag header that IIS 7 has made it impossible to remove via any configuration.

Removing ETag is one of the best ways to get better caching from browser for the static files on your website. Here's a HTTP Module that can remove the headers that you don't need:

```
public class RemoveASPNETStuff : IHttpModule
{
public void Init(HttpApplication app)
{
    app.PostReleaseRequestState += app_
PostReleaseRequestState;
}

void app_PostReleaseRequestState(object sender, EventArgs
e)
{     var headers = HttpContext.Current.Response.Headers;
    headers.Remove("Server");
    headers.Remove("X-AspNet-Version");
    headers.Remove("ETag");
}
```

Before the HttpModule:

Cache-Control:private
Content-Length:445
Content-Type:text/html; charset=utf-8
Date:Sun, 31 Mar 2013 11:19:36 GMT
Server:Microsoft-IIS/8.0
Vary:Accept-Encoding
X-AspNet-Version:4.0.30319

After the HttpModule:

Cache-Control:private
Content-Length:445
Content-Type:text/html; charset=utf-8
Date:Sun, 31 Mar 2013 11:16:03 GMT
Vary:Accept-Encoding

06 SYNCHRONIZING FILE DATE TIME ACROSS MULTIPLE SERVERS

When you deploy the same website on multiple webservers, you end up having each file getting different last modified date. As a result, each IIS (on different servers) produces different ETag for the static files. If a user is hitting different servers for the same file (due to load balancing), each IIS is responding with a different ETag and thus the browser downloading the same file over and over again. If you have 3 servers, the same user has most likely downloaded the same file thrice. This gives poor page load performance.

Moreover, if you want to mirror two or more locations using one location as a base, not only do you need to copy the same files, but you also need to set the same Create Date and Last Modified Date on the files. Otherwise they aren't true mirrors. There are various use cases where you need a complete mirror not only at file content level, but also at file date time level.

Here's a powershell script that will do the job for you:

```
# Path of the base folder. File date times in this folder
is used as the base.
$SourceFolder = ".Folder1"
# Put all the other locations here. These locations must
have the same folder structure as the base
$DestFolders = @('.Folder2', '.Folder3')

function sync($sourcePath, $destinationPath)
{
    $sourceFiles = [System.IO.Directory]::GetFiles(
                                    $sourcePath);
    foreach ($sourceFileName in $sourceFiles)
    {
        $sourceFile = Get-Item $sourceFileName
        $destFilePath = Join-Path -Path $destinationPath
        -ChildPath $sourceFile.Name
        $destFile = Get-Item $destFilePath
        if ($destFile.Length -eq $sourceFile.Length)
        {
            $destFile.LastWriteTime = $sourceFile.
                                    LastWriteTime;
            $destFile.CreationTime = $sourceFile.
                                    CreationTime;

            Write-Host ("SYNCED: " + $sourceFileName + "
                        -> " + $destinationPath)
        }
        else
        {
            Write-Host ("SIZE DOES NOT MATCH: " +
            $sourceFileName + " -> " + $destinationPath)
        }
    }

    $childFolders = [System.IO.Directory]::GetDirectories
                    ($sourcePath);
    foreach ($childFolderName in $childFolders)
    {
        $childFolder = Get-Item $childFolderName
        $destFolderPath = Join-Path -Path
        $destinationPath -ChildPath $childFolder.Name
        $destFolder = Get-Item $destFolderPath
        sync $childFolder.FullName $destFolder.FullName
    }
}

$Source = Get-Item $SourceFolder
foreach ($destFolderName in $DestFolders)
{
    $destFolder = Get-Item $destFolderName
    sync $Source.FullName $destFolder.FullName
}
```

07 AUTOMATE PRODUCTION DEPLOYMENT WITH SCRIPTS

When you have multiple webservers running your website, it becomes difficult and time consuming to deploy code on each server manually. You have to take one server out of the load balancer at a time, deploy the new code, warm it up, and put it back in to the load balancer. Then take another server out and keep repeating the steps until all the webservers have the latest code. There are always chances of human errors screwing up one server, which becomes very difficult to figure out once the site is up and running and you hear users reporting random problems.

Here's a scripted approach we take to deploy code on production:

```
@echo off

echo *** Taking server out of rotation ***
ren alive.txt dead.txt
typeperf "\W3SVC_W3WP(_Total)\Requests / Sec" -sc 10

echo *** Start deploying files... ***

echo *** Copy website files... ***

echo *** Run database upgrade scripts... ***

echo *** Files deployed... ***
ping -n 5 127.0.0.1 > nul

echo *** Bringing server back online... ***
ren dead.txt alive.txt

typeperf "\W3SVC_W3WP(_Total)\Requests / Sec" -sc 10
```

Figure 4 Deploy code on production via script

This is not a complete script for deploying sites on production, but a concept. I want to show you the ideas behind such automation. You need to add real code to deploy your website's codebase, as it suits you.

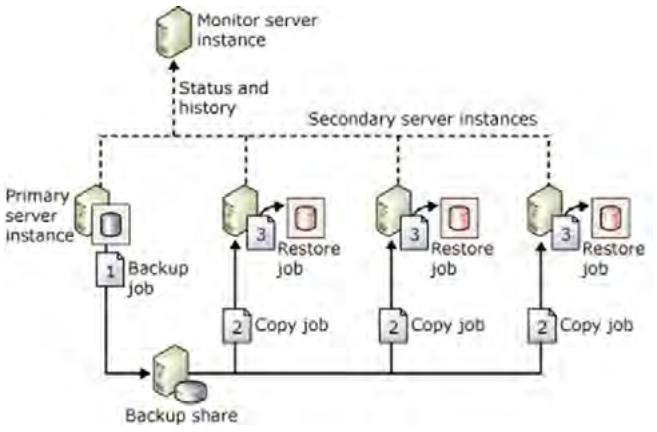
First, there's a file on the website called alive.txt. We configure our load balancer to hit this file every second to make sure the webserver is alive. If Load Balancer gets a HTTP 200, then it assumes the website is up and running. But if it gets some other response, then it assumes the site is down and it takes the webserver out of the rotation. It stops sending any more traffic to it, until it gets the alive.txt response.

So during deployment, we rename this file to say dead.txt so that load balancer stops sending traffic to this webserver. Then we give it some time to drain out the existing traffic. When the server has finished running all the in-flight requests, we see Requests/Sec showing as 0 from the typeperf command. Then the code deployment starts. There are various ways to do it. You can have the production binaries stored in a network share and then do a xcopy from that share into the server. Or you can have network binaries stored in subversion and download the binaries from subversion.

Then the script looks for .sql files in a specific folder and it runs them in proper order. We name the .sql files as 001, 002, 003 so that the exact sequence is maintained. The script connects to SQL Server using osql and executes the sql scripts one by one. Once the deployment is over, it renames the dead.txt back to alive.txt. Then load balancer detects the file is available again and it puts the webserver back into rotation.

08 CREATE READONLY COPY OF DATABASES USING SQL SERVER LOG

SQL Server Log shipping is a very easy to use technology to maintain a standby database server in order to survive a server failure. But you can use log shipping to create readonly copy of databases where you can divert readonly queries. This is a great way to scale out your database. You don't have to rely on only one server to serve all database traffic. You can distribute traffic across multiple servers.



Here you have a primary server where all the INSERT, UPDATE, DELETE happens. Then the database is copied to several other servers via log shipping, where a readonly copy of the database is maintained. You can configure log shipping to ship changes made in the primary database every minute or every 15 mins.

All the other servers have the database in readonly mode and it can serve SELECT queries. If you have an ecommerce website, then you have a product catalog that changes not so often. You can then divert all queries to the catalog tables to the log shipped servers and distribute the database load.

You should never produce MIS reports or take batch extracts from the primary database where the INSERT, UPDATE, DELETE are happening. You should always do these on log shipped standby databases.

09 HOUSEKEEPING DATABASES

If you have transactional tables where records get inserted very frequently and unless you remove older records from the table, you run out of space within weeks; then it is a challenge to periodically purge such tables, without bringing down the tables. If you have a 24x7 website, then it becomes very difficult to find a slot where you can take an outage and do a purge that takes on an average of 15-30 mins to finish. During purge operations, the table gets locked and all other queries on that table times out. So, you need to find a way to purge records consuming the least amount of database resource.

There are various ways to do this but we found the following approach having the least footprint on the database:

```
Declare @batchsize int; Set @batchsize = 1000;

set transaction isolation level read uncommitted; set
nocount on;

declare @temp_rows_to_delete table (ID int primary key)
insert into @temp_rows_to_delete select ID FROM
VeryLargeTable WHERE ID < 4000

Declare @rowCount int; Set @rowCount = 1;
declare @BatchIds table (id int primary key)

While @rowCount > 0
Begin
-- Pick the IDs to delete and remove them from #temp_rows
delete top(@batchsize) from @temp_rows_to_delete
                OUTPUT deleted.ID into @BatchIds
delete from VeryLargeTable WHERE ID IN (SELECT ID FROM @
BatchIds)
delete from @BatchIds
    Set @rowCount = @@rowCount;
Print 'Deleted ' + convert(varchar(10), @rowcount) + '
rows...'
    waitfor delay '00:00:05'
End;
go
Checkpoint 30;  -- simple recovery, if full - put tlog
backup here
Go
```

The idea here is to do a steady small batch of delete at a time until all the records we want to purge are deleted. This query can go on for hours, not a problem. But during that time, there will be no significant stress on the database to cause other queries to timeout or degrade significantly.

First we set the isolation level. This is absolutely a key thing to do. This isolation level tells SQL Server that the following query does not need locking. It can read dirty data, does not matter. So, there's no reason for SQL Server to lock records in order to ensure the query gets properly committed data. Unless you are a bank, you should put this in all your stored procs and you will see 10 times better throughput sometimes from your database, for read queries.

Next we read the row IDs from the source table and store in a table variable. This is the only time we are going to do a SELECT on the source table so that SQL Server does not have to scan the large table again and again.

Then we keep picking 1000 row IDs at a time and delete the rows from the table. After each delete, we give SQL Server 5 seconds to rest and flush the logs.

10 TUNING DATABASE INDEXES

Over the time, data in tables get fragmented as data gets inserted, updated and deleted. If you leave the tables alone, the indexes will get slower and slower resulting in slower queries. If you have millions of records on a table and you start having significant fragmentation on the tables, over the time, queries will get so slow that they will start timing out.

In order to keep the tables fast and lean, you need to regularly rebuild the indexes. Here's a SQL snippet that will run through all the indexes on the database and see if the indexes are heavily fragmented. If they are, then it will issue an index rebuild on them.

Ref: http://www.mssqltips.com/sqlservertip/1165/managing-sql-server-database-fragmentation/

When you run the query, it looks like this:

```
C:\>osql -S .\sqlexpress -d DatabaseName -E -i index.sql
1> 2> 3> 4> 5> 6> 7> 8> 9> 10> 11> 12> 13> 14> 15> 16> 17> 18> 19> 20> 21> 22> 2
3> 24> 25> 26> 27> 28> 29> 30> 31> 32> 33> 34> 35> 36> 37> 38> 39> 40> 41> 42> 4
3> 44> 45> 46> 47> 48> 49> 50> 51> 52> 53> 54> 55> 56> 57> 58> 59> 60> 61> 62> 6
3> 64> 65> 66> 67> 68> 69> 70> 71> 72> 73> 74> 75> 76> 77> 78> 79> 80> 81> 82> 8
3> 84> 85> 86> 87> 88> 89> 90> 91> 92> 93> 94> 95> 96> 97> 98> 99> Executing DBCC
DBREINDEX (aspnet_Users,
aspnet_Users_Index) - fragmentation currently 97%
DBCC DBREINDEX (aspnet_Users,aspnet_Users_Index)
Executing DBCC DBREINDEX (aspnet_Users,PK__aspnet_U__1788CC4D20C1E124)
- fragmentation currently 95%
DBCC DBREINDEX (aspnet_Users,PK__aspnet_U__1788CC4D20C1E124)
Executing DBCC DBREINDEX (aspnet_Users,aspnet_Users_Index2)
- fragmentation currently 44%
```

You should do this during weekends when your traffic is lowest. However, there's a catch. SQL Server Standard Edition does not allow online index rebuild. This means during index rebuild, the index will be offline. So, most of your queries will timeout. If you want to have the index online during the rebuild, then you will need SQL Server Enterprise Edition, which is expensive if you are on a tight budget.

11 MONITORING SQL SERVER FOR PERFORMANCE ISSUES

Windows Performance Monitor is a great way to monitor performance of your databases. Just go to Start > Run, type "perfmon" and you will get the performance monitor. From there, click on the + sign and add the following counters:

MSSQL$SQLEXPRESS:Access Methods	
Full Scans/sec	30.000
Page Splits/sec	4.000
MSSQL$SQLEXPRESS:Databases	**_Total**
Transactions/sec	95.999
MSSQL$SQLEXPRESS:Locks	**_Total**
Average Wait Time (ms)	60.429
Lock Requests/sec	184,013.907
Lock Timeouts/sec	169.998
Lock Wait Time (ms)	422.995
Lock Waits/sec	7.000
MSSQL$SQLEXPRESS:SQL Statistics	
Batch Requests/sec	22.000
Processor Information	**_Total**
% Processor Time	85.960

These are some key counters that can give you an indication about how well your database is performing and if there's a performance problem, what is the bottleneck.

For example, here you see very high Full Scans/Sec. This means you have queries that are not using index and scanning the entire table.

So, you need to investigate which queries are causing high IO and see their execution plan to find out where they are missing index. You can run SQL Server Standard Reports by right clicking on the server on SQL Server Management Studio and selecting Reports and then the Top 10 IO and Top 10 CPU consuming queries. They will show you the worst performing queries. That's your starting point to fine tune.

The other most significant one is Lock Requests/Sec. It should ideally be zero. But if you have more than 100, then it is usually bad. It is an indication that you need to put SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED in your queries.

12 SENDING MAILS FROM WEBSITES

We usually use the SmtpClient to send emails via some SMTP server. Usually you have a common email server for your company and your websites connect to that email server to send mails. But if you configure SmtpClient to connect to the mail server, then it is going to open a connection to the email server every time you call the Send() function. Establishing a SMTP session, uploading the email message, then the attachments is a very chatty process. There are many back and forth communications that happens between the webserver and email server. It makes your pages to slow down when you try to synchronously send email. If your SMTP server goes down, your website throws error and the messages are lost.

Instead of configuring SMTP server settings in web.config, you should configure to use the local IIS SMTP Service. On each webserver, install the IIS SMTP Service and configure it to relay the messages to the company's email server. This way your website will connect to the local server and queue the messages into the IIS SMTP service. Since this process happens within the server, it will be a lot faster than connecting to a distant email server. IIS SMTP service will gradually pick the messages and send them to the company's email server.

An even better approach is to use the IIS SMTP service pickup

folder. SmtpClient can just serialize the whole message in a pickup folder usually located at C:\INETPUB\MAILROOT\PICKUP. This way even if the IIT SMTP Service is down, it will still be able to write the message and complete the execution of the page. Then IIS SMTP service will pick up the messages from this folder and send them one by one.

All you have to do is have this in your web.config:

```
<system.net>
    <mailSettings>
      <smtp deliveryMethod="PickupDirectoryFromIis" />
    </mailSettings>
  </system.net>
```

13 REGULAR SERVER RESTART

Sometimes we have bad code on production that causes memory leaks, crashes the app pool randomly, allocates COM Objects but does not release them gracefully, opens TCP connections and does not close them properly. This causes things to go bad at operating system level and you end up with an unhealthy server. You start having weird problems and crashes. Sometimes IIS stops responding. Sometimes remote desktop does not work. At that point, you have no choice but to restart Windows.

Especially if you outsource development to cheapest possible resource, you will have no choice but to regularly restart servers. We have seen such problems occurring so many times that nowadays we regularly restart windows servers at least once every quarter.

14 DISKCLEANUP

Windows downloads gigantic windows updates, creates gigabytes of crash dump when app pool crashes, fills up temporary folders with gigabytes of garbage and so on. If you have limited storage on your webservers, for example if you are on those cheap virtual servers that comes with 20 GB space, then you need to schedule regular diskcleanup.

Fortunately Windows comes with a pretty decent cleanup tool. First you run: `Cleanmgr /sageset:###`

Put some number there. It will run Disk Cleanup tool in a record mode where it will let you select what you want to clean. Then when you click OK and exit, it will remember those settings against that number.

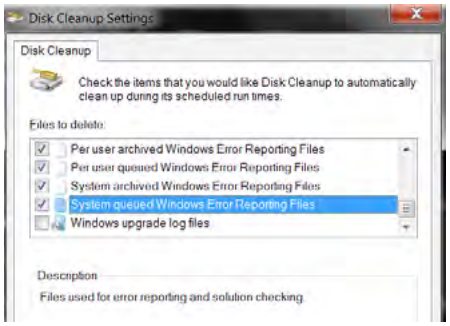
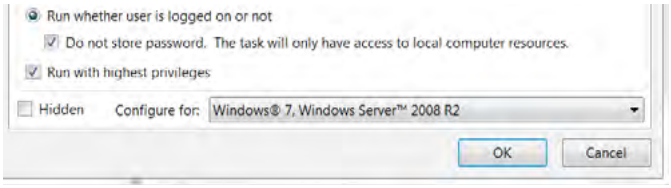


Figure: Diskcleanup in record mode

Then you schedule a task in Windows Task Scheduler to run the following command:

```
Cleanmgr /sagerun:###
```

Here you put the same number that you have used to record the settings. When you configure the task, ensure you have these settings turned on:



This will give Cleanmgr the necessary privilege to delete files from protected folders. ■

Omar is the Chief Architect of SaaS Platform at BT in London, UK. He is a Microsoft MVP, CodeProject MVP and author of an O'Reilly title - "Building Web 2.0 Portal with ASP.NET 3.5". Omar has several popular Open Source projects including Dropthings, CodeUML, PlantUMLEditor. Omar's specialization on Performance and Scalability techniques can be found in his blog – http://omaralzabir.com

REAL-TIME NETWORK APPLICATIONS IN .NET

MS MVP, *Filip Ekberg* explains the difference between real-time and near real-time systems. He then shows us how to build a near real-time system using the .NET SignalR framework.

In the real world, there are mentions of real-time systems all over the place. These systems are often defined as something that needs to respond in just a couple of milliseconds, irrespective of the load. Even a couple of milliseconds in some cases is not acceptable because real-time ideally is and should be, *latency free*.

This is really hard to achieve depending on the type of system that you are working on. In most cases, real-time systems have operations that are more prioritized than others. For example think of a military fighter jet. If you're in one of these and you really need to eject your seat, a latency of two seconds would really not be acceptable. These type of systems have certain exceptions or interrupts that override normal flow of operations. These interrupt requests thus go in front of anything else and are processed almost as soon as they occur.

As you might have guessed, these type of real-time systems are not created using a managed language such as C# mostly because of memory management (garbage collection), that introduces an 'unpredictable' latency. Though, intelligent buffering can mitigate some of the latency to achieve what is commonly referred to as "near real-time" responses in a system; C# still isn't choice of language for mission-critical real-time systems.

Figure 1

Distributed components and "near Real-Time" applications in .NET

Now that we've gotten the definition of a real-time system out of the way, what type of near real-time applications can we create in .NET? The perception from end users is that when something responds fast, it's real-time. There's another factor that matters here as well and that is how long does our data travel and how many places is it distributed to.

"Near Real-Time" responses in Web Applications

Let's say a web application is running in a geographically close vicinity (say same Network or even same City) and you ask it to process something. If it returns results in a few hundred milliseconds or thereabouts, it is considered near real-time. But if it takes more than a couple of hundred milliseconds, it loses the 'perception' of near real-time. On the contrary, if the request was sent off to a server in Japan and you're located in France where the result was returned in say five hundred milliseconds instead of two hundred, it would still be considered fast and near real-time. Now let's add another parameter to this and introduce multiple interactions. This can be multiple users using and processing data at the same time, where all the data is presented to every interaction point. One example of such interaction is Google Docs feature where it lets you create and edit text documents simultaneously with your peers.

As you might understand, these aren't true real-time systems but the perception of instant updates makes it feel very much so. There are parameters that we cannot control such as network latency, disconnects from other clients and so forth.

These end up in actual real-time systems as well, but in those applications, the components are usually much closer together.

Real-time means different things to different people, but now that we have a common ground to stand on, let's head on and look at how we did this prior to today in .NET.

REAL-TIME NETWORK APPLICATIONS IN THE PAST USING .NET

There are multiple ways to solve different scenarios regarding real-time systems using .NET. You might have found yourself using raw sockets to have full control over everything you do. The biggest problem is that if you are creating a real-time application using .NET and rely on sockets to help you, you most likely write a nice wrapper or library that you can use.

As with anything else, this ends up in lots and lots of different libraries to solve real-time systems over sockets. There's certainly nothing wrong with going down on the lowest possible layer, but sometimes abstractions can reduce complexity and speed up development.

If you want to handle disconnects, reconnects and broadcasts, you'll first and foremost need a basic list of all connected and possibly disconnected clients as you can see above. There's absolutely nothing wrong with this, but so far we've just started to talk about the most basic things. Imagine what this adds up to once the real-time system grows.

Next up, you need a unified language that all your clients and servers use, so that you can ensure that each entry point gets what they should.

The following code shows how to create a basic server that listens on TCP for clients to connect and once they connect, it simply sends a message to the client greeting it.

```
var socket = new Socket(SocketType.Stream, ProtocolType.
Tcp);
socket.Bind(new IPEndPoint(IPAddress.Parse("127.0.0.1"),
3425));
socket.Listen(5000);
while (true)
{
 var client = socket.Accept();
 Console.WriteLine("Client connected!");
 client.Send(Encoding.Default.GetBytes("Hello!"));
}
```

Next up, we need to connect to this server and to do this, we have the following code.

```
var client = new Socket(SocketType.Stream, ProtocolType.
Tcp);
client.Connect(new IPEndPoint(IPAddress.
Parse("127.0.0.1"), 3425));
var buffer = new byte[1024];
while (true)
{
 Console.WriteLine("Connected to server!");
 client.Receive(buffer);
 Console.WriteLine(Encoding.Default.GetString(buffer));
}
```

As you might have noticed, there are a couple of problems with the server and the client. First of all, the buffer is big and we don't handle data sent which is more than 1024 bytes. This would need to be fixed if this was a real world application. Writing up all this boilerplate code for a real-time application, can be cumbersome and we tend to do the same thing over and over again. The next step when we have a client and a server, is to come up with a "protocol" to use and now I am *not* talking about the actually DTOs that are sent, but in what way they are sent.

This all adds up and in the end you will end up with something that someone else has also written lots of times before. Therefore it is now time to present the solution of repetitive boilerplate for near real-time communication between Client and Server – a.k.a. **SignalR.**

CREATING REAL-TIME NETWORK APPLICATIONS WITH SIGNALR

SignalR is a real-time network library that has an awesome and well-engineered abstraction over sockets. Instead of having to write all the socket code yourself, you use SignalR to fire up a server which is as simple as saying the port number out loud, a couple of times. Then you have very useful features that will let you direct messages to one client, multiple clients or to just broadcast the entire message.

In SignalR, you have abstractions that will even help you to do things even more magically. This actually means that there is a "raw" way for working with SignalR and there's a more easy way to work with clients and the server.

SignalR has fall back mechanisms built in on different protocols and technologies. For instance, if you set up a server, and the client only has support for long polling; that is what is going to be used.

WHAT IS LONG POLLING

This technique is the most basic low level one that will "always" work and the theory behind it, is as follows. When a client connects, it asks for a message from the server. If the server responds, then the client initiates the connection again until a response is received. But if there is no response sent from the server and the request times out, the client will then reconnect and continue asking for a response. This is called long polling because the client polls the server for answers all the time and the response timeout is long.

There are other newer technologies that can be used, such as WebSockets; this will however only be used when the host and the client has support for it. The host? SignalR can be hosted in multiple ways. The most common way is to host SignalR in an ASP.NET application, but you can also self-host SignalR with *OWIN*.

As you might have understood, SignalR can be used exactly as you would use sockets, but since the data is sent in a uniform way, the client and server will always understand each other. This makes it much easier to develop with.

Basically what you can do is to set up different messages that the server understands and then once these messages are received from the clients, certain actions are invoked. Think of this as events that you trigger on the server from the client.

As visualized in the figure, a message is sent from a client to a server and then this message triggers an action, or an event per se, which in return sends a message to the client with a response. This response message could have been broadcasted to all the connected clients, one client or multiple clients.

You might ask yourself, how can you distinguish between different clients and this is a very good question. With each connection, you get an ID that identifies the current connection. This identification number, which really is a Guid is used when you send a message to the client or a bunch of clients.

It is good practice to save each connection id that is connected and somehow identify each connection as an individual. This could be done by asking to authenticate when starting the communication; authentication such as using Twitter or Facebook. These authentications could then on multiple occasions be used to tie a connection id to an actual individual. We will look more at how the connection id and information regarding it is processed in the next sections.

With an understanding of what SignalR is, we can go on and look at the different possibilities that we have to host servers and connect with clients.

Raw (Persistent) connections

The raw connections in SignalR is pretty much as working with sockets, but with a very nice abstraction on top of sockets. What it really means is that it is the lowest abstraction level inside SignalR that you can work with and it gives you as much power as if you were working directly with sockets.

In order to get started, we need to fire up a new project and setup SignalR. First off, create a new ASP.NET Empty Web Application. Once this project is created, open up the package manager console (NuGet) to write the following:

```
Install-Package Microsoft.AspNet.SignalR
```

This will install whatever is needed to setup our project as a SignalR based server. To setup a class to use as a server, create a new file called MyServer and add the following to the file:

```
using System;
using System.Linq;
using Microsoft.AspNet.SignalR;

public class MyServer : PersistentConnection
{
}
```

Inheriting from Persistent connection allows us to override some very useful methods. Before we do that though, we need to configure our application to know that this class is going to be our server. Create the file global.asax and add the following:

```
public class Global : System.Web.HttpApplication
{
 protected void Application_Start(object sender,
EventArgs e)
 {
  RouteTable.Routes.MapConnection<MyServer>("", "/");
 }
}
```

Now we can start adding things to the server. Open up MyServer again and override a method called OnConnected. This method will be invoked each time a client connects to the server. These methods will all return *Task* so you can work with this in an asynchronous manner.

The first parameter gives us information about the current request such as the user, data posted and things like that. The second parameter identifies the connection based on a connection id (Guid). This id can be used to direct messages to the client.

There's a property in PersistentConnection called Connection that we can use to either broadcast or send data to all or one client. If we want to broadcast to everyone except certain clients, we can do this as well because Broadcast takes a second parameter which tells us which clients to exclude.

In order to broadcast a message to each client that connects, we do the following:

```
protected override Task OnConnected(IRequest request,
string
connectionId)
{
 return Connection.Broadcast("Hello there!");
}
```

The same goes for sending a message to a specific client except Send takes another parameter that defines what client is receiving the message. As you might understand, this is quite low level, and there's no protocol defined for how the messages will look or anything like that. You'll need to figure that out yourself when using persistent connections. So far, it's still a lot nicer than working directly with sockets. It's time to start the project, as you might see it will throw an error because your web browser does not speak the language the server understands, so just ignore this for now.

Connection to the server

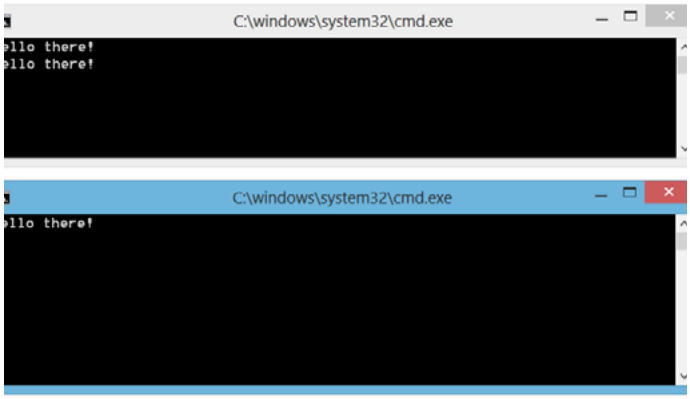
To test if the message really is broadcasted to all clients, we need to setup a couple of clients. Create a new console application called MyClient and fire up the package management console again and write the following:

```
Install-Package Microsoft.AspNet.SignalR.Client
```

Now open up Program.cs and add the following to your main method:

```
var myServer = new Connection("http://localhost:8082/");
myServer.Received += (string data) => { Console.
WriteLine(data); };
myServer.Start();
```

You might need to change the port for your SignalR server. Once you've done that, fire up a couple of instances of this application. As you can see, the first one will get as many greetings as the number of clients. This is because we are using broadcasting to send the message to all the clients, as you can see below



You'll notice that each message is sent instantly to each client making it feel very close to real-time. Remember though that this is on a local system and that we do indeed have some overhead in a real-world scenario.

Persistent connections give you great power but you're not getting much more from the framework other than a nice abstraction on top of the underlying transportation. As mentioned previously, SignalR will detect which transportation is most suitable and use that.

Hubs

If you're not too comfortable working with something raw as persistent connection, you really don't have to worry. With SignalR comes another abstraction called Hubs. This is an abstraction on top of Persistent Connections which will make everything a lot easier. *If it wasn't already easy enough, right?*

With Persistent Connections and larger applications, you might imagine that you'd have to create a message loop which translates the data sent to the server and calls the correct delegate for that event. With Hubs, this is completely done for you. You can simply create methods that you register to certain events and are raised by clients.

To set it up, create a new empty web application project called HubsServer and install SignalR into it using the package management console as you saw before.

Create a new class called MyHubServer and inherit from Hub. This will let you, pretty much like we saw with Persistent Connection,

override methods such as OnConnect, OnDisconnect and OnReconnect. We will leave these out of the picture this time though. First let us define what the hub is going to do; in this case we want it to handle adding and retrieving information about customers. Therefore we will have three methods with the following signatures:

- void Add(Customer customer)
- IEnumerable<Customer> All()
- Customer Get(int id)

As you can see, these look like they could be a part of any customer handling API out there. The customer entity is very simple, all it has is a name and an id which we store in an in-memory list. This should end up looking like the following:

```
using Microsoft.AspNet.SignalR;
using System;
using System.Collections.Generic;
using System.Linq;

namespace HubsServer
{
 public class MyHubServer : Hub
 {
  public static IList<Customer> Customers { get; set; }
  public void Add(Customer customer)
  {
   if (Customers == null) { Customers = new
                            List<Customer>(); }
   Customers.Add(customer);
  }
  public IEnumerable<Customer> All()
  {
   return Customers;
  }
  public Customer Get(int id)
  {
   if (Customers == null) return null;
   return Customers.FirstOrDefault(x => x.Id == id);
  }
 }

 public class Customer
 {
  public int Id { get; set; }
  public string Name { get; set; }
 }
}
```

Before we can test this out, we need to tell our application to register the SignalR hubs in the application. This is done by adding the following to Global.asax:

```
using System;
using System.Linq;
using System.Web.Routing;

namespace HubsServer
{
 public class Global : System.Web.HttpApplication
 {
  protected void Application_Start(object sender,
EventArgs e)
  {
   RouteTable.Routes.MapHubs();
  }
 }
}
```

If you run this by hitting F5 in Visual Studio, it will bring up a browser for you. Ignore the 403 Error that you get and append the following to the URL: /signalr/hubs

This will show you a generated JavaScript that SignalR has prepared for us and if you scroll down to the bottom of the script, you'll see something very interesting:

```
proxies.myHubServer.server = {
 add: function (customer) {
  return proxies.myHubServer.invoke.apply(proxies.myHub-
Server, $.merge(["Add"],
   $.makeArray(arguments)));
 },

 all: function () {
  return proxies.myHubServer.invoke.apply(proxies.myHub-
Server, $.merge(["All"],
   $.makeArray(arguments)));
 },

 get: function (id) {
  return proxies.myHubServer.invoke.apply(proxies.myHub-
Server, $.merge(["Get"],
   $.makeArray(arguments)));
 }
};
```

As you can see, there are three definitions for methods that exists on the server and the signatures corresponds with what we created in our C# class! Another very neat thing here is that it changed the method names to camel case to correspond with the JavaScript coding guidelines.

Connecting to the Hub

We can try this out in the same application as we have the server running, by adding a new HTML page for adding and displaying customers. Add a file called Client.html and add the following to the header section to include the SignalR generated JavaScript, jQuery and the SignalR common JavaScript:

```
<script src="Scripts/jquery-1.6.4.min.js"></script>
<script src="Scripts/jquery.signalR-1.0.1.min.js">
</script>
<script src="/signalr/hubs"></script>
```

The actual form will look very simple, it will only have two fields and two buttons like you can see below in figure 5

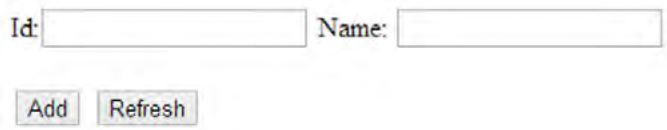


Figure 5

The HTML itself is equally simple:

```
<body>
 Id:<input type="text" id="id" name="id" />
 Name: <input type="text" id="name" name="name" />
 <br /><br />
 <input type="button" value="Add" id="add" name="add" />
 <input type="button" value="Refresh" id="refresh"
name="refresh" />
 <div id="result" />
</body>
```

When the document has finished loading, we want to hook up SignalR and perform some magic. First we get a reference to myHubServer through the $.connection object like this:

```
var myHub = $.connection.myHubServer;
```

We can then use myHub to send and receive messages from the server. For instance, in order to call all() we simply do this:

```
myHub.server.all();
```

There's a common pattern here so we can append the method done() after our calls, meaning that once all is finished, we can execute a method with the result data. To do this and then add a new list with the items returned from the server to the page, we can simply do the following:

```
$("#refresh").click(function () {
 myHub.server.all().done(function (result) {
  var resultDiv = $("#result");
  resultDiv.html("");
  resultDiv.append("<ul>");
  $(result).each(function (index, item) {
   resultDiv.append("<li>("+item.Id+") "+ item.Name +
    "</li>");
  });
  resultDiv.append("</ul>");
 })
});
```

This hooks up a click handler to the refresh button and if we have any elements in the customer array, it will be displayed nicely on the page. In order to test that, we need to add some elements. This can be done in an equal manner by running the add() method we created before and passing it a JSON object like this:

```
$("#add").click(function () {
 myHub.server.add({ id: $("#id").val(), name: $("#name").
val() });
});
```

Last but not least, we can initiate the connection to the server:

```
$.connection.hub.start();
```

Run this and 'Add' some names and click on Refresh. This will give us an archaic looking list as we can see in Figure 6.

Notice what happens if you open up a new tab in your browser, navigate to the same html file and hit refresh, you'll get a list of all the names! Aside from the network latency and other factors, this is so far a one-way-push-real-time network application!

What about if we want to notify the clients once a new customer has been added and refresh automatically? We could do this quite easily. Let's start off by adding an event handler on the client side called refreshCustomers like this:

```
myHub.client.refreshCustomers = function (result) {
 // Same code here as refresh click handler
};
```

Then on the server side, all we have to do is call this method using the dynamic property *All* on the Clients property. Clients lets us direct messages to one or many clients and in this case we will run the event on each client and pass the corresponding data to them:

```
Clients.All.RefreshCustomers(Customers);
```

The property *All* is dynamic and we can run the methods that we think are on the client and if they are not present, there won't be any invocation. We're now as close to a real-time network application as we can be as we have the push to server and push to clients.

It was previously mentioned that this could have been any customer handling system out there and as you've seen, it could as well have been the customer model in our application that we pass around. We wouldn't really need to do any changes to those since SignalR supports complex DTOs.

Connecting to the Hub in a .NET Console Application

We've now seen how we can create a web-browser based client but what if we want to use this in our .NET Applications that do not run in the browser? Let's take a look at something similar to what we did with Persistent Connection.

Create a new Console Application called MyHubClient and install the SignalR client using the package management console like this:

```
Install-Package Microsoft.AspNet.SignalR.Client
```

The code that follows is very similar to what we saw in JavaScript. We first create a connection to where the SignalR server is and then we create a proxy for the hub that we are going to use like this:

```
var connection = new HubConnection("http://local-
host:3448/");
var myHub = connection.CreateHubProxy("MyHubServer");
```

This is equivalent to doing $.connection.myHubServer in JavaScript.

Next up we're ready to invoke the methods. This client will ask for a specific customer id and list that id. In order to invoke a method on the server, there's a method on the myHub object that is called Invoke. All these methods are asynchronous and await able, precisely as they are in JavaScript as well.

As the entry point of the application cannot be marked as asynchronous, we need to move this to a separate method, to keep it simple:

```
using Microsoft.AspNet.SignalR.Client.Hubs;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace MyHubClient
{
 class Program
 {
  static void Main(string[] args)
  {
   RunHubClient();
   while (true) ;
  }

  static async Task RunHubClient()
  {
   var connection = new HubConnection("
                        http://localhost:3448/");
   var myHub = connection.CreateHubProxy("MyHubServer");
   // Connect to the hub
   connection.Start();
  }
 }
}
```

The infinite while loop after the method call is just there so that the application won't exit. Now in order to ask for a specific customer, we could use the generic Invoke method and tell it that we will receive a dynamic object like this:

```
var customer = await myHub.Invoke<dynamic>("Get",
customerId);
```

This will invoke the method Get on the server passing the value of whatever customerId read from the console. When the invocation is done, it will return to the continuation block beneath it and execute whatever code is there.

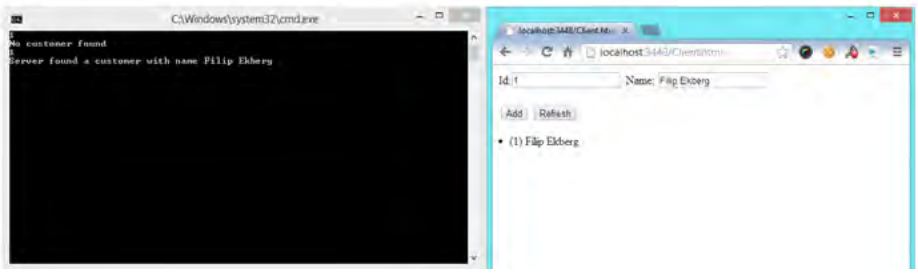
This method can end up looking like this:

```
static async Task RunHubClient()
{
 var connection = new HubConnection("http://local-
host:3448/");
 var myHub = connection.CreateHubProxy("MyHubServer");
 connection.Start();
 string line = null;
 int customerId = 0;
```

```
 while ((line = Console.ReadLine()) != null)
 {
  if (int.TryParse(line, out customerId))
  {
   var customer = await myHub.Invoke<dynamic>("Get",
                                              customerId);
   if (customer == null) Console.WriteLine("No customer
                                              found");
   else Console.WriteLine("Server found a customer with
                                 name {0}", customer.Name);
  }
 }
}
```

Because marking void methods asynchronous is a very bad practice, we've set it to return a task so that we can follow the operation if we'd like to. The method will read a line from the command line and try to parse it as an integer, then it will either tell us the customer name or tell us that there are no customers at all.

Fire up both the console application and the website and first ask for a customer before you've added a customer, then add a customer in the web interface and ask again as you can see in the Figure below:



We'd also like the console application to subscribe to the event where the client is notified when a customer is added. This is easily done through the hub proxy just as we did with the invocation of a method, but this time we use the method called *On* instead. This method defines that we are listening for a certain message. The first parameter is the message that we are listening to and the second is the action to run when the message is received.

This needs to be added before we start the connection and the code looks like this:

```
myHub.On("RefreshCustomers", (customers) => {
    Console.WriteLine("New customers added, total amount
of customers {0}", customers.Count);
});
```

If we run this as you see in Figure 8 and add some customers in the web interface, we see that the console application is notified when a new customer is added and then prints the amount of customers. The count is taken from the parameter that is sent to the action while the data is passed all the way from the SignalR hub.

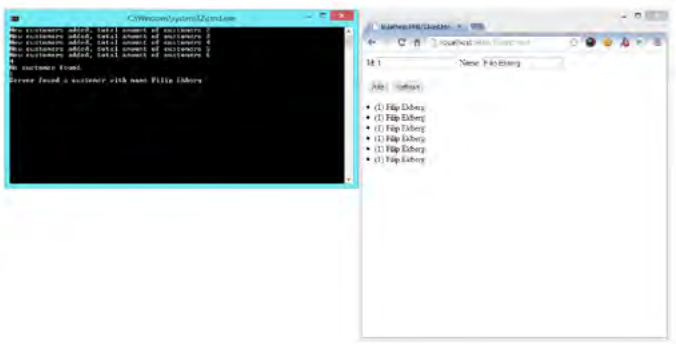


Figure 8

Connecting to the Hub in WinRT using HTML & JavaScript

We've seen how to connect to hubs using a web based client as well as a normal .NET application running in Windows, but what about Windows 8, more specifically WinRT? Unfortunately it's not as simple as just including the JavaScript on the server in a WinRT application because WinRT Applications don't allow loading external resources. But it's not too difficult either.

First off create a new blank JavaScript Windows Store application, you can call this MyWinRTHubClient. We can't simply install SignalR using NuGet this time, instead we need to manually add some necessary files. We need the following files for it to work, these can be copied from the server project we created earlier:

- jquery.signalR-1.0.1.min.js
- jquery-1.6.4.min.js

Copy these files over to the js directory in the Windows Store project and reference them in the default HTML file created for the project. The JavaScript that we're adding is really not that different from what we saw in the normal web-browser client; we can even use the same HTML!

When the document has finished loading, we want to create a hub connection and create a proxy for the SignalR server. This looks similar to what we saw before, but notice that it's not entirely the same:

```
var connection = $.hubConnection('http://local-
host:3448/');
```

```
connection.start();

var myHub = connection.createHubProxy("myHubServer");
```

The JavaScript to listen for events and to invoke methods look much more similar to what we saw in the normal console application rather than what we saw in the earlier web client. To invoke a method, we simply call myHub.invoke(methodName, value) and it will look like this:

```
myHub.invoke("add", ({ id: $("#id").val(), name:
$("#name").val() }));
```

Compare this to what we saw earlier in the web-browser client:

```
myHub.server.add({ id: $("#id").val(), name: $("#name").
val() });
```

Same goes for listening for events, we can simply use myHub.

```
on(methodName, action) like this:
myHub.on('refreshCustomers',function (result) {
 refreshCustomers(result);
});
```

This means that there were actually very few changes in the WinRT version in order to get this to work properly. Check the source code at the end of this article to see and compare the entire html file in the WinRT application.

Running this code in the simulator is similar to what you can see in the figure below. Just as you might imagine, the refresh, add and event, works just as expected.

CONCLUSION

While "real" real-time network applications goes much deeper than what we've explored now; you should now have a good feeling about how you could tackle something that should feel as near as real time as possible with as little hassle as possible. We've seen how we can use SignalR to create low level messaging channels using Persistent Connections and then move over to the abstraction on top of it using Hubs to create much nicer interactions between the servers and the clients.

Using SignalR truly makes life easier when needing to receive, push or subscribe to data/messages on a remote server and if you've done network programming earlier, you certainly know that this comes as a blessing ■

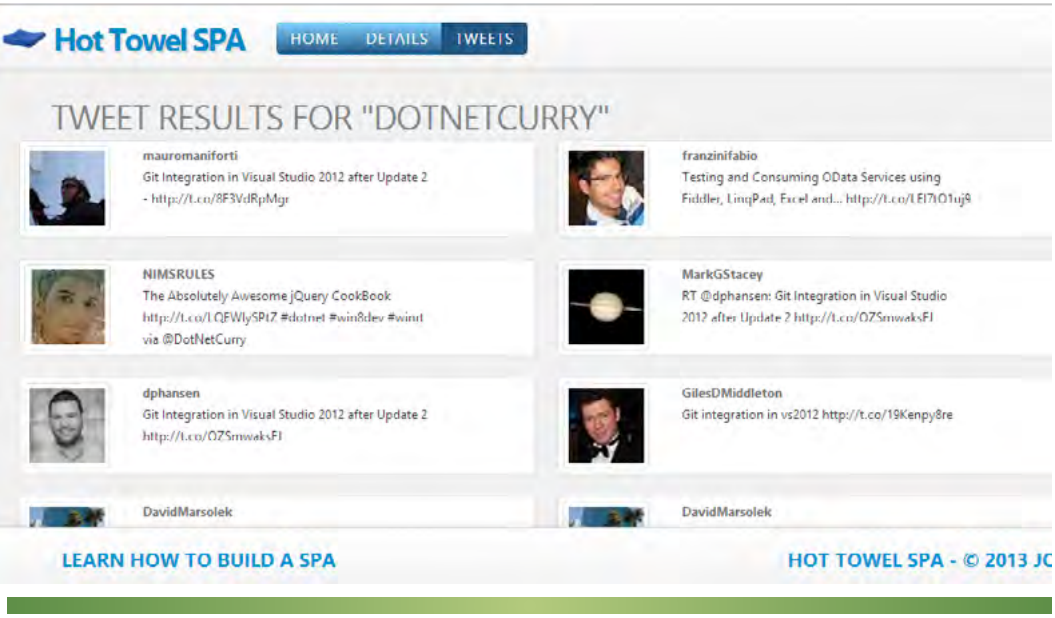
Download the entire source code from Github at bit.ly/dncm6-siglr

Filip is a Microsoft Visual C# MVP and a Senior Software Engineer with a big heart for programming in C#. Most recently, Filip is the author of the book C# Smorgasbord, which covers a vast variety of different technologies, patterns & practices. You can get a copy of the ebook for only €4.99 by following this link: http://shorturl.se/?smorgasbord. Follow Filip on twitter @fekberg and read his articles on fekberg.com


Join us on
Facebook
www.facebook.com/
dotnetcurry

REMEMBER YOUR **HOT TOWEL** WHEN GOING TO THE **SPA**

Technology evangelist *Fanie Reynders* pokes around in Hot Towel - a Single Page Application template in ASP.NET MVC 4; and shows how easy it is to kick start your next project for the modern web.



> Single Page Applications (SPAs) allows us to have the fluid user experience of a desktop app, together with the scalability of a web app

Ultimately, the goal of any app is to give the user the best experience possible. The way old-fashioned websites work, causes disruption in this user experience flow. Performing a simple task with traditional web apps causes the entire page to post-back to the server, have a cup of coffee and return back to the client with (hopefully) a different screen.

Single Page Applications (SPAs) are one of the latest trends in web technology today. It allows us to have the fluid user experience of a desktop app together with the scalability of a web app

GETTING STARTED

Bundled with the ASP.NET and Web Tools 2012.2 Update is a new SPA template that allows developers to build rich interactive client-side web apps using HTML5, CSS3 and some JavaScript libraries. The ASP.NET team provided one template and the Community jumped in with five more.

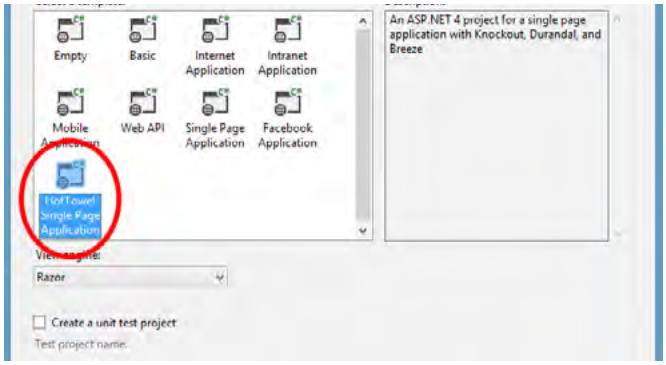
One of the community driven templates you can use is Hot Towel. It was built by John Papa. It comes power-packed with everything you need out of the box and provides you with modular code structure, powerful data management and easy elegant styling. The template gives you everything you need to get started with SPA so you can focus on your app and not the "other stuff". Download the VSIX here: bit.ly/151Mk9o (direct download).

Pre-requisites

- Visual Studio 2012 or Visual Studio Express 2012 for Web
- ASP.NET Web Tools 2012.2 update

Let's do this, shall we?

In Microsoft Visual Studio 2012, select 'File' > 'New' > 'Project...' and then pick the 'ASP.NET MVC 4 Web Application' template from the Web section. Name the project something interesting and click 'OK'. You will be presented by the following New MVC 4 Project dialog:



The dialog above may be familiar to those of you that are well versed in ASP.NET MVC, but after installing the ASP.NET Web Tools 2012.2 update, you will notice a new template has been added to the list: 'HotTowel Single Page Application'. Pick it and click 'OK'.

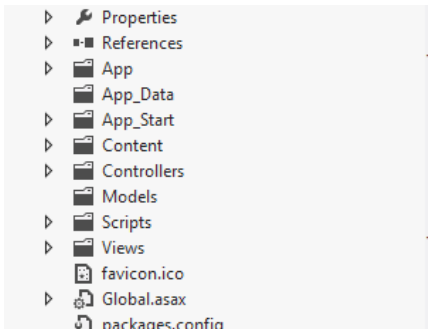
The SPA HotTowel template is also available on NuGet:

```
PM> Install-Package HotTowel
```

You can find out more by heading here:
http://nuget.org/packages/HotTowel/

Poking around the structure

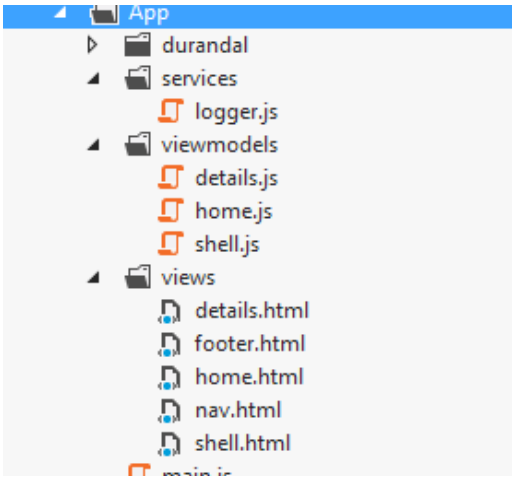
After Visual Studio did its thing, you will notice that the project has a similar structure than the good old stock standard ASP.NET MVC 4 project.



This includes:

- App_Start – Start-up server side logic
- Content – Usually contains images and css
- Controllers – Page & API Controllers
- Models – Model & View Model classes
- Scripts – JavaScript files
- Views – All the pages for the application

In addition, we notice that HotTowel SPA provides an 'App' folder which contains a collection of modules that encapsulates functionality and declares dependencies on other modules. Here is where all the magic happens. Your entire application basically lives in this domain.



Durandal is a cross-device, cross-platform client framework and designed to make SPAs easy to create.

The presentation logic for the views resides in the 'viewmodels' folder. This is a common MVVM pattern.

In the 'views' folder, you will see pure HTML files for your application. This is mainly driven by templating functionality of Knockout (a JavaScript library for MVVM).

View by feature

The SPA HotTowel template comes power-packed with the following functionality:

- ASP.NET MVC
- ASP.NET Web API
- ASP.NET Web Optimization for bundling and minification
- Breeze.js for rich data management
- Durandal.js for view composition, app life cycle and navigation
- Knockout.js for data bindings

- Require.js for modularity
- Toastr.js for notifications (pop-up messages)
- Twitter Bootstrap for styling

4 Easy steps to get building on HotTowel

1. Add your custom server-side code (preferably ASP.NET Web API and/or Entity Framework)
2. Create your viewmodels in the 'App/viewmodels' folder
3. Create your screens (views) using HTML in the 'App/views' folder and implement the Knockout data bindings in your views
4. Update the navigation routes in 'shell.js'

WALKTHROUGH

Index.cshtml

Looking in the 'Views/HotTowel' folder, we find a view called 'Index.cshtml' that serves as the starting point for the MVC application. In a SPA (as the name implies), we are only using one page to host a collection of views for our application. The 'Index.cshtml' view contains all the meta-tags, style sheet and JavaScript references you would expect. Let's examine the 'body' tag:

```
<div id="applicationHost">
 @Html.Partial("_splash")
</div>

@Scripts.Render("~/scripts/vendor")
<script type="text/javascript"
src="~/App/durandal/amd/require.js"
data-main="@Url.Content("~/App/main")"></script>
```

The 'applicationHost' div is the host for the HTML views. Next it renders the 'vendor' bundle that contains all the external vendor JavaScript references.

The entrance point for the application's code is specified by referencing Require.js and assigning the 'main' data-attribute to find the start-up JavaScript logic.

Main.js

This the start-up code for your application and contains the navigation routes, the start-up view definition and any bootstrap logic you may want to implement when the application initially fires up.

Several of Durandal's modules are defined in Main.js to help kick start the process, the 'define' statement helps resolve module dependencies so they are available within the context.

```
define(['durandal/app', 'durandal/viewLocator',
'durandal/system', 'durandal/plugins/router',
'services/logger'],
 function (app, viewLocator, system, router, logger)
{
  // Enable debug message to show in the console
  system.debug(true);
  app.start().then(function () {
   toastr.options.positionClass =
                              'toast-bottom-right';
   toastr.options.backgroundpositionClass =
                              'toast-bottom-right';
   router.handleInvalidRoute = function (route,
    params) {
    logger.logError('No Route Found', route, 'main',
     true);
   };

   // When finding a viewmodel module, replace the
   // viewmodel string with view to find it partner
   //  view.
   router.useConvention();
   viewLocator.useConvention();
   // Adapt to touch devices
   app.adaptToDevice();
   //Show the app by setting the root view model for
   // our application.
   app.setRoot('viewmodels/shell', 'entrance');
  });
});
```

Views

All the views are found in the 'App/views' folder and only contains pure HTML files. The 'shell.html' contains the master layout of our application (similar to the '_Layout.cshtml') and all of the other views will be composed somewhere inside this.

3 regions are used in the layout view: A header, content area (section) and footer. Each of these regions will be loaded with content from other views when requested.

```
<div>
 <header>
```

```
  <!--ko compose: {view: 'nav'} --><!--/ko-->
 </header>
 <section id="content" class="main container-fluid">
  <!--ko compose: {model: router.activeItem,
   afterCompose: router.afterCompose,
   transition: 'entrance'} -->
  <!--/ko-->
 </section>
 <footer>
  <!--ko compose: {view: 'footer'} --><!--/ko-->
 </footer>
</div>
```

In 'nav.html' is an example of how Knockout.js is leveraged for rich model binding:

```
<nav class="navbar navbar-fixed-top">
 <div class="navbar-inner">
  <a class="brand" href="/">
   <span class="title">Hot Towel SPA</span>
  </a>
  <div class="btn-group" data-bind="foreach: router.
      visibleRoutes">
   <a data-bind="css: { active: isActive }, attr: {
    href: hash }, text: name"
    class="btn btn-info" href="#"></a>
  </div>
  <div class="loader pull-right" data-bind="css: {
    active: router.isNavigating }">
   <div class="progress progress-striped active
         page-progress-bar">
    <div class="bar" style="width: 100px;"></div>
   </div>
  </div>
 </div>
</nav>
```

ViewModels

The application's viewmodels are found in the 'App/viewmodels' folder. Inside the 'shell.js', 'home.js' and 'details.js' are properties and functions bound to 'shell.html', 'home.html' and 'details.html' respectively. For example, let's explore 'home.html':

```
<section>
 <h2 class="page-title" data-bind="text: title"></h2>
</section>
```

And 'home.js':

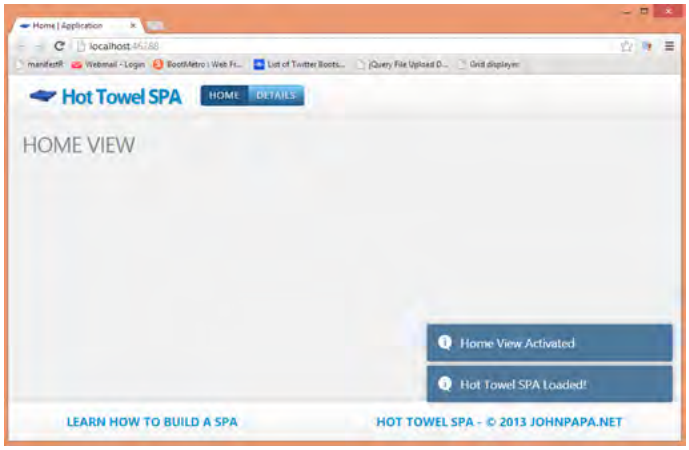
```
define(['services/logger'], function (logger) {
 var vm = {
  activate: activate,
  title: 'Home View'
 };
 return vm;

 function activate() {
  logger.log('Home View Activated', null, 'home', true);
  return true;
 }
});
```

Services

All the application specific services are found in the 'App/services' folder. HotTowel comes standard with the 'logger.js' service responsible for logging messages and also displaying it on the screen as toast notifications. The 'services' folder is the ideal place to put logic like data-retrieval for instance.

RUNNING THE TEMPLATE

Out of the box, HotTowel is ready to run. Just hit F5 and you'll get:



HANDS ON EXAMPLE

Let's say you want to create a page called 'Tweets' that shows the latest tweets from DotNetCurry.

Implement the service

We will need some kind of service that will be responsible for retrieving tweets from Twitter, given a certain search term. Right-click the 'App/services' folder and add a new JavaScript file called 'twitter.js' with the following contents:

```
define(function () {
  var tweetModel = function (user, image, tweet) {
  this.user = user;
   this.tweet = tweet;
   this.image = image;
  };
  var model = {
   loadTweets: load,
  };
  return model;

  function load(query, list) {
   return $.ajax({
    url: "http://search.twitter.com/search.json?q=" +
          query,
    crossDomain: true,
    dataType: "jsonp"
   }).done(function (data) {
    $.each(data.results, function (i, d) {
    list.push(new tweetModel(d.from_user,
        d.profile_image_url, d.text));
   });
  });
 }
});
```

The service exposes a function called 'loadTweets' that expects a query and a list object. Given the parameters, it will asynchronously load tweets with the given search query and populate the list object which must be observable by Knockout.

Create the ViewModel

Now we need to consume our Twitter service. Right-click the 'App/viewmodels' folder and add a new JavaScript called 'tweets.js' with the following contents:

```
define(['services/logger', 'services/twitter'], function
(logger, twitter) {
 var vm = {
  activate: activate,
  title: 'Tweet results for "DotNetCurry"',
  tweets: ko.observableArray([])
 };
 return vm;

 function activate() {
  twitter.loadTweets("dotnetcurry", vm.tweets);
  logger.log('Tweet page activated', null, 'tweets',
true);
  return true;
 }
});
```

Notice how we resolve dependencies using the 'define' function. In this case, I need to use the logger service and our twitter service situated in 'services/logger' and 'services/twitter' respectively.

One of the properties the model comprises of, is 'tweets'. Notice that it is an observable array, which means Knockout will be handling the real-time change tracking.

On the model activation, we use the Twitter service to load tweets with query "dotnetcurry" as well as passing in the observed tweet list object (from our model).

Create the View

Now for the Tweet page. Right click the 'App/views' folder and add a HTML page with the name 'tweets.html' containing the following HTML:

```
<section>
 <h2 class="page-title" data-bind="text: title"></h2>
 <ul class="thumbnails" data-bind="foreach: tweets">
  <li class="span5">
   <div class="tweet row thumbnail">
  <img class="span1 thumbnail" data-bind="attr: { src:
image }" />
    <div class="span3">
     <span class="twitter-user" data-bind="text:
user"></span>
     <span data-bind="text: tweet"></span>
    </div>
   </div>
  </li>
 </ul>
</section>
```

For each tweet in the model, it will create a list item tag containing an image of the user including his/her name as well as the tweet that was posted.

*Note that I am using Bootstrap's styling to accomplish the nice effects as well as my own custom styles.

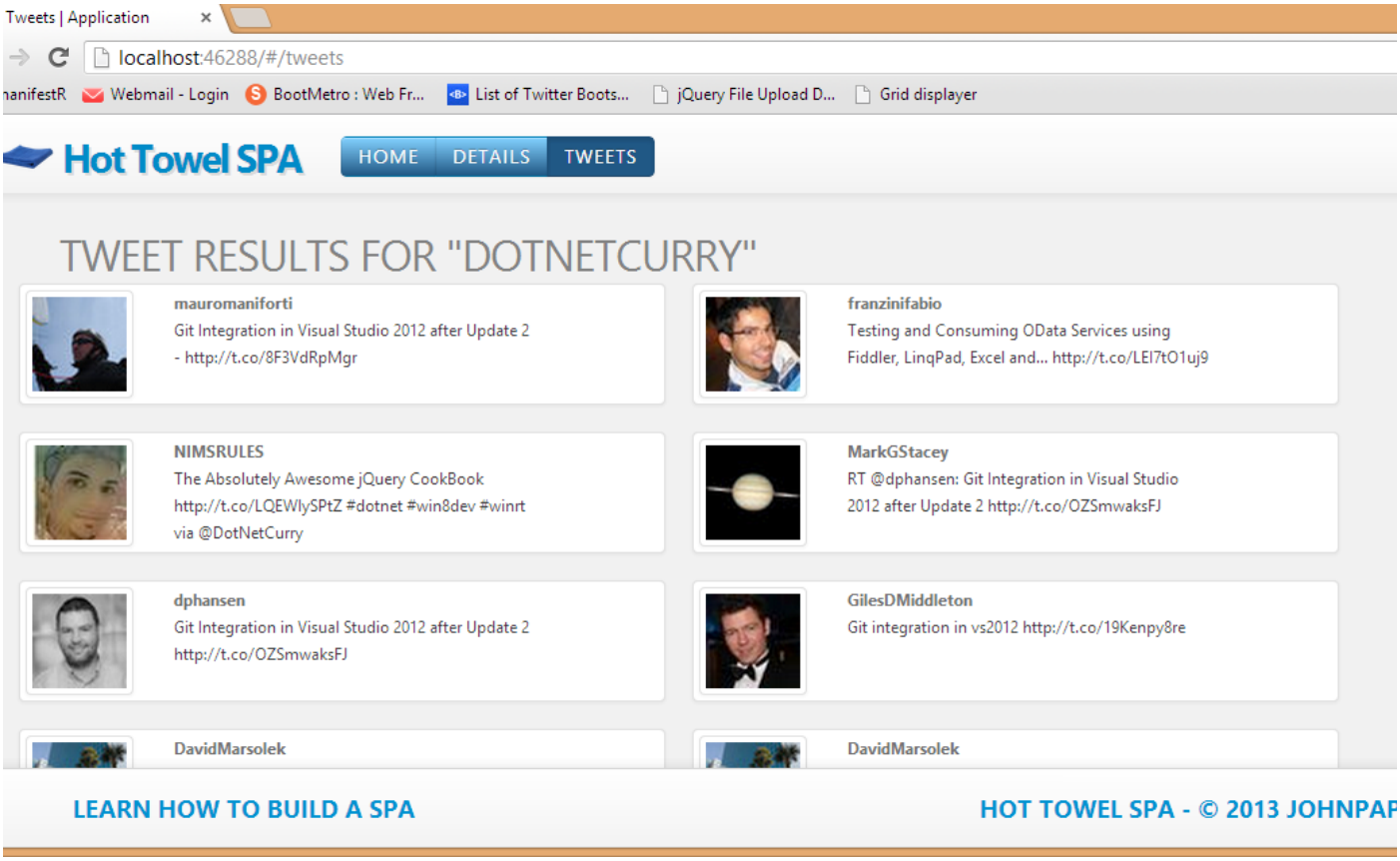
Wire up the navigation

The last thing we must do is to tell the shell about the new page. In the 'App/viewmodels/shell.js' file add the map of the tweet page on the boot function:

```
function boot() {
 router.mapNav('home');
 router.mapNav('details');
 router.mapNav('tweets');
 log('Hot Towel SPA Loaded!', null, true);
 return router.activate('home');
}
```

LET IT RUN!

After hitting F5, you'll notice a new menu item 'tweets'. After clicking it, we are presented with the tweet page that we just created:



CONCLUSION

The power of SPAs are great and endless and the new HotTowel SPA template is a great way to get started. As application developers, we don't want to worry about the boring plumbing but be able to quickly start working with the fun stuff. This template allows us to do just that!

HotTowel really makes it easy to create fast, durable, modular, cross-browser and sexy Single Page Applications with rich navigation and data access functionality. This is definitely on the tech-stack for my next web app ■

You can download the entire code from Github here bit.ly/dncm6-ht

Fanie Reynders is a Microsoft Certified Professional from South Africa. Being a web technology evangelist in his own right, he has been developing on the Microsoft stack for over 9 years. Follow him on Twitter @FanieReynders and read his blog on bit.ly/ZiKLT5

DOCUMENT DRIVEN ALM USING TFS APIs

VS ALM MVP *Subodh Sohoni* explains how to use TFS APIs in collaboration with the SharePoint platform, so that documents drive the Application Lifecycle Management processes.

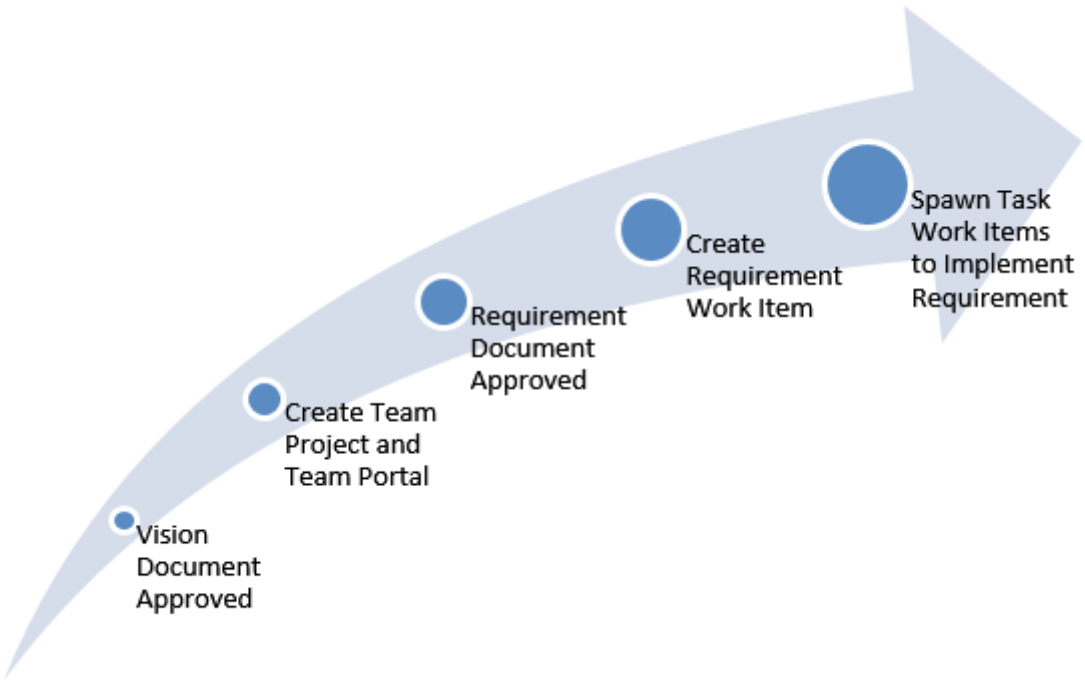
TFS IS AWESOME

Microsoft Team Foundation Server is one of the best products for Application Lifecycle Management. It does an excellent job of planning, monitoring, tracking and controlling various artifacts that we come across, while managing the application development. It treats all those artifacts as work items that are records in SQL Server and stores the data and metadata of those artifacts. For example, it stores the User Story as descriptive narration of a scenario, which is the data part of that user story and it also stores the status of that user story, which is the metadata part of the same user story. This schema gives us a unified view of these artifacts like user stories, requirements, tasks, risks, issues, test cases, bugs and many more. It is common to find projects that are driven by these artifacts in the form of work items. Change of the metadata of an artifact like status of a work item from New – Active – Resolved – Closed moves the project forward in small steps.

At this point now, we must accept that although work items conveniently provide unified view of the artifacts, it is also common practice in the industry to store the data in the form of document at some other repository and store only the metadata in TFS. For example, requirements in details may be stored in the "Requirements Document" whereas each requirement may have an associated work item of type Requirement or User Story or PBI to it. Traditionally, people find it easier to work with documents, edit them with the help of excellent tools like MS Office and store them in a repository like a file system or SharePoint Libraries. Many a times, I have seen projects in which the application development is not driven by work items, but are driven by the documents themselves stored outside TFS.

Typically, a project starts when a Vision document is created and approved. That document may include a Vision Statement, overall scope of the project and the organization of the team. Once this document is approved, the project kick starts. The Team then collects the requirements and a requirement document is created. The raw requirements may come in the form of Minutes of the Meeting, Emails or a document created by the customer. This requirement document goes through a process of refinements and approvals. At the end of that process, it is finalized so that initial requirements / user stories / PBIs can be added in the TFS. A link back to the requirement document is inserted in each of these work items. A similar process is carried out for architecture of the solution where design documents and diagrams are the driving force. Approval of these architectural artifacts triggers the construction (development) phase of the project. We may assume that if the team follows one of the Agile methodologies, these architectural documents may not exist, but they do exist to a certain extent. Iteration zero of any project that follows Agile creates an overall architectural strategy for the solution and that is documented. Any phase change in the project is documented regardless of methodology that is being followed.

We will take a case where the documents are stored in SharePoint list that works as the document repository. TFS stores the metadata in the form of work items for the same documents. These work items can be used for creating hierarchies and for the purpose of reporting. What we would like to achieve, is a workflow like this:



1. When a Vision document is approved in the SharePoint list, trigger the process to create a new team project in TFS. Name of the team project should be given as a property of the document when it is approved.

2. Link another list of documents to an approval workflow, so that a document like a requirement document (SRS) is created – reviewed – approved by the authorized team members.

3. When a document is finalized, appropriate work items should get created with the required metadata in the appropriate team project in TFS.

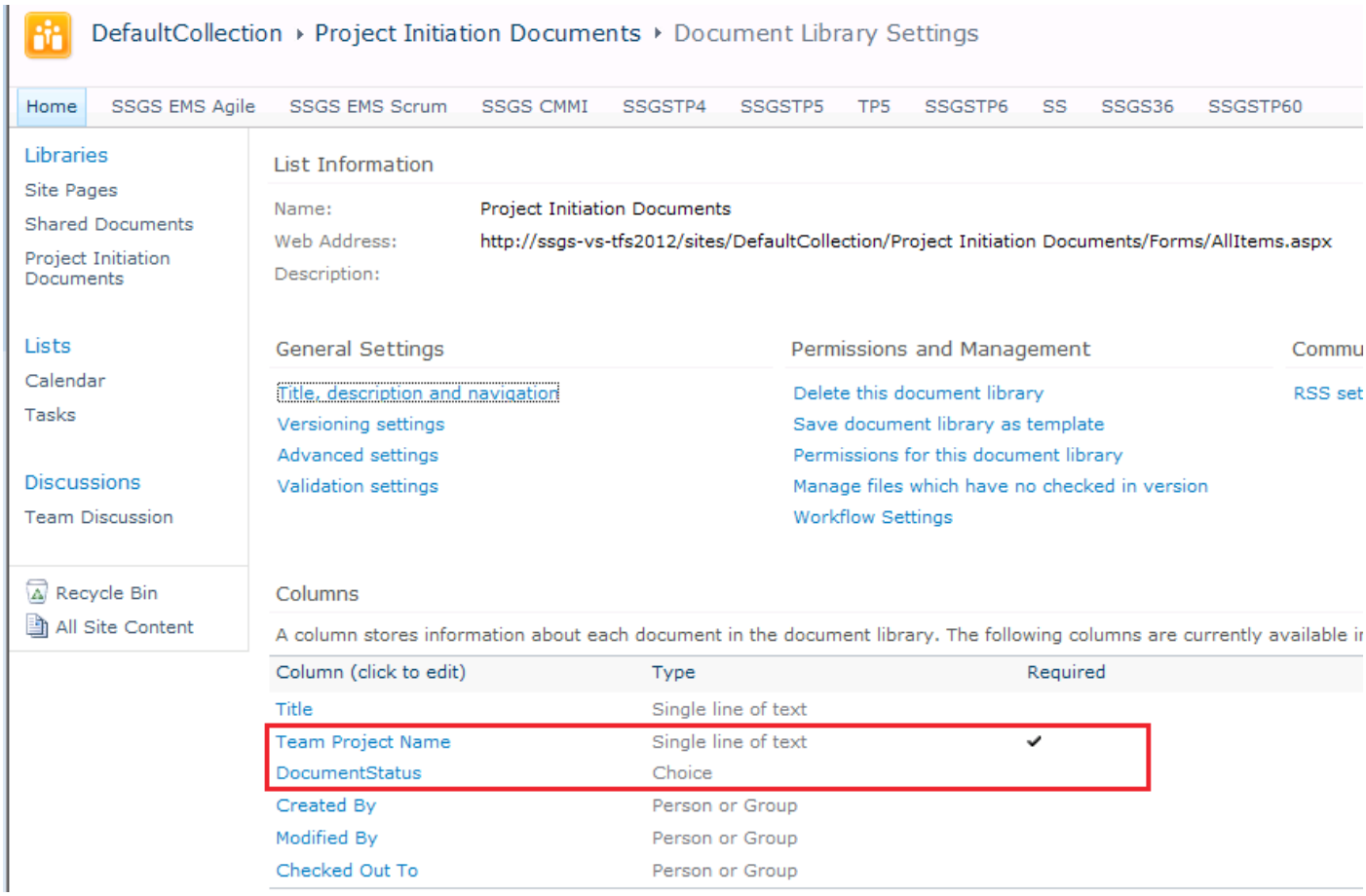
4. When a requirement is added in the team project, a set of tasks should get spawned to implement that requirement.

In this case, we will first focus on the vision document and then a requirement document as the driver. Processing the requirement document creates an additional complexity that is, a requirement document may (and usually does) contain multiple requirements. We will need to read those requirements and create appropriate work items for each requirement. This will require that the structure of the requirement document should be predefined using a template. This type of template is a common practice.

We begin by creating a library of documents in the portal that is at the team project collection level. This library will be used to host the 'Vision document'. We name that document library as 'Project Initiation Documents' and keep a template of such documents in the library. Permissions to this library are set in such a way that only the people who are involved in the project creation can access it. We create two columns for this library using 'Library Settings' page. One of them is a simple one line text property named as 'Team Project Name'. This property allows the initiator to assign a name to the team project. Second column represents status of the document which can be

- Needs Approval
- Approved
- Needs Rework

For this coulmn, we select the type as Choice and provide these values as available options.



Now we will create a workflow in Visual Studio 2012. To do so, we select the project type 'SharePoint 2010 Project'. We create this project to program a workflow that will give us a chance to run our code when the Vision document is approved by the authorized person. We will not go in too many details of how to create a workflow (You can find a detailed walkthrough of that http://msdn.microsoft.com/en-us/library/ee231573.aspx). We will focus on the code that accesses TFS to create a new team project. We will write that code in the event handler that handles the event of onWorkflowItemChanged. As the name suggests, the event is raised when any change in the item that is hosted on our target SharePoint server, is detected. In this event handler, we will start an external process that will call a command to create a Team Project, with the name that is provided as a property of the Vision document when it is approved. Such a command can be written using a tool that is provided in TFS 2012 Power Tools. The command line utility that is part of these power tools is TFPT.exe. We can run the command

```
TFPT.exe createteamproject
```

This command accepts the following parameters:

/collection: URL of the Team Project Collection (e.g. http://TFSName:8080/tfs/CollectionName)

/teamproject: Team Project Name

/processtemplate: Name of the process template that is already uploaded to TFS (e.g. "MSF for Agile Software Development 6.0")

/sourcecontrol: New or None or branch:BranchPath

/noreports: If reports are not desired in the team project

/noportal: If the portal is not desired for the team project

First three parameters are compulsory and the remaining are optional in nature. Example of a complete command may look like this:

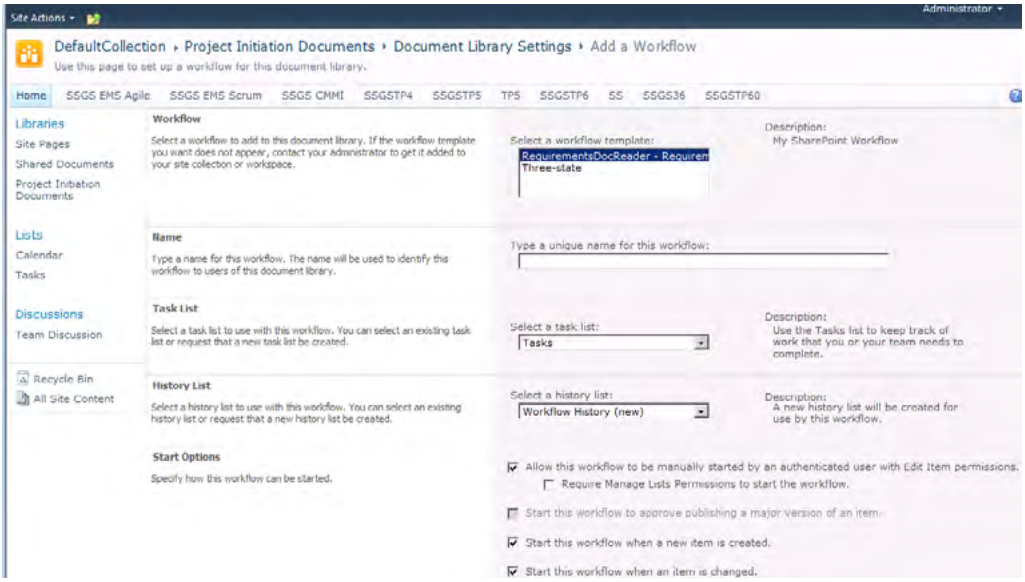
```
TFPT.exe createteamproject /collection:http://SSGS-TFS2012:8080/tfs/DefaultCollection /teamproject:"SSGS EMS Agile" /processteamplate: "MSF for Agile Software Development 6.0" /sourcecontrol:New /noreports / noportal /verbose /log:"C:\temp\logs"
```

Code for that event handler will be:

```
private void CreateTeamProject(string ProjectName)
{
    string args = "tfpt.exe
```

```
createteamproject /collection:http://ssgs-vs-tfs2012:8080/
tfs/DefaultCollection /teamproject:\"" + ProjectName + "\"
/processtemplate:\"MSF for CMMI Process Improvement 6.2
\"";
    ProcessStartInfo psi =new ProcessStartInfo("cmd.exe");
    psi.UseShellExecute = false;
    psi.RedirectStandardOutput = true;
    psi.RedirectStandardInput = true;
    psi.RedirectStandardError = true;
    psi.WorkingDirectory = @"C:\";
    psi.Verb = "runas";
    //start the process
    Process p = Process.Start(psi);
    p.StandardInput.WriteLine(args);
    p.WaitForExit();
    p.StandardInput.WriteLine("EXIT");
    p.Close();
}
```

This workflow project is built and deployed to the SharePoint server. After deployment, it is associated with the document library that is hosting the Vision Document. It is configured to run when a Vision document is created and then changed having the state 'Approved'. This workflow will trigger the creation of a team project. The name of that team project is provided by the approver as property of the vision document in the SharePoint Library.



Now we will focus on another driving document – Requirement Document. We create a document library in the team portal that will provide a repository for requirement document for the project. We will call that document library as 'Requirement Documents'. The requirement document is a standardized document in every organization and its structure does not change much. We are going to assume that it is the MS Excel worksheet that contains multiple requirements, each individually on a row and the title of each requirement is in column B.

For the document library, we will have a similar set of properties that we had created earlier for the library that we had created for Vision Document. We will have 'Team Project Name' and status as two properties.

We will also create a SharePoint workflow that will have the event handling code. In this code, we will read the requirement document and create work items of the type Requirement. First we should add a few references to the TFS API class libraries and their related using statements. We are going to use OpenXML that is a standard used for editing MS Office files programmatically without installing MS Office, usually on the server. We can download V 2.0 (one that I used) of that from http://www.microsoft.com/en-in/download/details.aspx?id=5124.

```
using Microsoft.TeamFoundation.Client;
using Microsoft.TeamFoundation.WorkItemTracking.Client;
```

The boilerplate code for such a SharePoint workflow project is as follows:

```
private void CreateRequirements()
{
prj = (string)workflowProper-
ties.Item["Team Project Name"];
collection = new
TfsTeamProjectCollection(new
Uri("http://ssgs-
vs-tfs2012:8080/tfs/
DefaultCollection"));
store = collection.
GetService<WorkItemStore>();
SPWeb web = workflowProperties.
Web;
SPFile file = web.
GetFile(@"http://
ssgs-vs-tfs2012/sites/
DefaultCollection/SSGS%20CMMI/
Requirements%20Documents/
RequirementDocument.xlsx");
SPFileStream dataStream = (SPFileStream)file.
OpenBinaryStream();
SpreadsheetDocument document = SpreadsheetDocument.
Open(dataStream, false);
WorkbookPart workbookPart = document.WorkbookPart;
IEnumerable<Sheet> sheets = document.WorkbookPart.
Workbook.GetFirstChild<Sheets>().Elements<Sheet>();
```

```
string relationshipId = sheets.First().Id.Value;
WorksheetPart worksheetPart = (WorksheetPart)document.
            WorkbookPart.GetPartById(relationshipId);
Worksheet workSheet = worksheetPart.Worksheet;
SheetData sheetData = workSheet.
GetFirstChild<SheetData>();

IEnumerable<Row> rows = sheetData.Descendants<Row>();
int i = rows.Count();
foreach (Cell cell in rows.ElementAt(0))
{
string Title = GetCellValue(document, cell);
WorkItem Requirement = new WorkItem(store.Projects[prj].
                        WorkItemTypes["Requirement"]);
Requirement.Title = Title;
Requirement.Save();
}

}
public static string GetCellValue(SpreadsheetDocument
document, Cell cell)
{
SharedStringTablePart stringTablePart = document.
             WorkbookPart.SharedStringTablePart;
string value = cell.CellValue.InnerXml;

if (cell.DataType != null && cell.DataType.Value ==
CellValues.SharedString)
{
   return stringTablePart.SharedStringTable.
       ChildElements[Int32.Parse(value)].InnerText;
}
else
{
return value;
}
}

private void checkStatus()
{
if ((string)workflowProperties.Item["Status"] ==
    "Approved")
       workflowPending = false;
}

private void onWorkflowActivated(object sender,
ExternalDataEventArgs e)
{
    checkStatus();
}

private void isWorkflowPending(object sender,
ConditionalEventArgs e)
{
    e.Result = workflowPending;
}

private void onWorkflowItemChanged(object sender,
ExternalDataEventArgs e)
{
  checkStatus();
  if ((string)workflowProperties.Item["Status"] ==
      "Approved")
{
    CreateRequirements();
}
}
```

This workflow is associated with the Document Library for Requirement Documents. Whenever the requirement document is approved, it will create the requirement work items in the team project.

Now we move our attention to the last part of the workflow. When a requirement is added in the team project, it should spawn some preconfigured tasks. These tasks should be linked to the same requirement as children. We can write an event handler for the event of 'WorkItemChanged' that is raised by TFS. It begins by creating a class library project. Add references to

- Microsoft.TeamFoundation.Client.dll
- Microsoft.TeamFoundation.Common.dll
- Microsoft.TeamFoundation.Framework.Server.dll
- Microsoft.TeamFoundaiton.WorkItemTracking.Client.dll
- Microsoft.TeamFoundaiton.WorkItemTracking.Server.
  Dataaccesslayer.dll

These references are located at

C:\Program Files(x86)\Microsoft Visual Studio 11\Common7\IDE\ ReferenceAssemblies\v2.0 and C:\Program Files\Microsoft Team Foundation Server 11.0\Application Tier\Web Services\bin folders

Code for the entire class that has the event handler method is as follows:

```
using System.Threading.Tasks;
using Microsoft.TeamFoundation.Client;
using Microsoft.TeamFoundation.WorkItemTracking.Client;
using Microsoft.TeamFoundation.Framework.Server;
```

```
using Microsoft.TeamFoundation.WorkItemTracking.Server;

namespace SpawnTasks
{
public class SpawnTasks: ISubscriber
{
public string Name
{
    get { return "WorkItemChangedEvent"; }
}

public SubscriberPriority Priority
{
    get { return SubscriberPriority.Normal; }
}

public EventNotificationStatus ProcessEvent(TeamFoundatio
nRequestContext requestContext, NotificationType
notificationType, object notificationEventArgs, out int
statusCode, out string statusMessage, out Microsoft.
TeamFoundation.Common.ExceptionPropertyCollection
properties)
{
    statusCode = 0;
    statusMessage = string.Empty;
    properties = null;
    try
    {
       if (notificationType == NotificationType.
       Notification && notificationEventArgs is
       WorkItemChangedEvent)
       {
           string machineName = System.Environment.
                                       MachineName;
           TfsTeamProjectCollection tpc = new
               TfsTeamProjectCollection(new Uri("http://"
         + machineName + ":8080/tfs/DefaultCollection"));
           WorkItemChangedEvent ev =
                notificationEventArgs
                                as WorkItemChangedEvent;
           WorkItemStore ws = tpc.
                          GetService<WorkItemStore>();
           Microsoft.TeamFoundation.WorkItemTracking.
            Client.WorkItem wi = ws.GetWorkItem(ev.
                 CoreFields.IntegerFields[0].NewValue);
           Project project = ws.Projects[ev.
                            PortfolioProject];
           if (wi.Type ==  project.WorkItemTypes["User
                    Story"] && wi.RelatedLinkCount == 0)
           {
                Microsoft.TeamFoundation.WorkItemTracking.
Client.WorkItem NewTask = new Microsoft.TeamFoundation.
WorkItemTracking.Client.WorkItem(project.
WorkItemTypes["Task"]);
                NewTask.Title = "Create architecture
                   diagrams for user story " + wi.Id;
                WorkItemLinkTypeEnd linkTypeEnd =
          ws.WorkItemLinkTypes.LinkTypeEnds["Parent"];
                NewTask.Links.Add(new
                      RelatedLink(linkTypeEnd,wi.Id));
                NewTask.Save();
           }
       }
    }
    catch
    {
    }
    return EventNotificationStatus.ActionApproved;
}

public Type[] SubscribedTypes()
{
    return new Type[] {typeof(WorkItemChangedEvent)};
}
}
}
```

Similar to creating one task as shown, it is possible to create multiple tasks and link all of them to the same requirement as children. After compilation, this assembly should be copied at C:\Program Files\Microsoft Team Foundation Server 11.0\Application Tier\Web Services\bin\plugins folder for it to work as event handler of TFS event of 'WorkItemChangedEvent'.

We have achieved the goal of driving the part of the ALM process through documents. Creation and approval of the workflows trigger the activities in TFS to create and update artifacts. This POC can be further extended to many other parts of the ALM like Test Case Management, bugs management (where a bug is filed by customer by email), change request management etc. ■

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

Interview with C# Legend
Eric Lippert

Dear Readers, we are very happy to have Eric Lippert to talk to in this episode of the DNC Magazine. Eric needs no introduction to the C# folks out there, but for others, Eric is well known for his work in the C# compiler team. He spent a significant part of his career at Microsoft in various roles. Prior to Microsoft, Eric worked for Watcom. The 'older' ones among us might remember Watcom as a company who made very good C++ and Fortran compilers. Currently he is working for Coverity helping build static code analysis products.

DNC: Welcome Eric, we are super excited to have you with us.
EL: Thanks, I'm happy to be here!

DNC: You worked at Microsoft for 16 long years. Describe your journey (if we may call it that) from internship to working on VBScript, JScript, VSTO and becoming a core member of the C# compiler team.
EL: I grew up in Waterloo; having an interest in science and mathematics from an early age, it was very natural for me to go to the University of Waterloo. And besides, I had relatives on the staff, I already knew a number of the professors, and as you said, I'd worked at UW spin-off company Watcom as a high school student, so I knew exactly what I was getting into. UW has an excellent co-operative education program - one of the largest in the world - through which I managed to get three internships at Microsoft working on the Visual Basic team. They were good enough to extend me a job offer when I graduated, and I stayed in the developer tools division for my entire career at Microsoft.

DNC: Since you started as an Intern at Microsoft, is it fair to assume you got a lot of good inputs from your seniors in the early days? What is the best programming advice you ever got?
EL: I got a lot of good advice from more senior engineers throughout my career, not just at the beginning; Microsoft encourages both formal and informal mentoring. I've talked about the best *career* advice I got at Microsoft elsewhere

recently: basically, it was to become a domain expert by answering as many user questions as I possibly could. But the best *programming* advice is a bit harder to put my finger on. I learned so much at Microsoft from world-class experts in programming language design, performance analysis and many other fields, that it is hard to name just one.

Here's one thing that has stuck with me from even before my Microsoft days though. One day many years ago Brian Kernighan gave a talk about programming at UW. At one point he put up a slide showing some code that was clearly wrong. You knew it was wrong because what the comment above the code said and what the code actually did were opposites. Kernighan said something to the effect of "pop quiz: what actually runs, the *code* or the *comments*?" I still ask myself that same rhetorical question when I'm trying to understand a piece of buggy code; it is frequently the case that the comments are misleading, out-of-date or just plain wrong, and it is frequently the case that the comments are right and you don't even bother to read the buggy code that follows them. Kernighan's talk changed my attitude towards commenting completely. Since then I've tried to write comments that explain what the *goal* of a particular piece of code is, rather than trying to explain *how it works*.

DNC: When your team started the development of C#, what were some key goals in your mind? Are you happy the way C# has matured?

EL: To be clear: I started working on C# just as the final corner cases in the design of C# 3.0 were being hammered out and development was beginning in earnest. I have followed C# carefully since its creation over a decade ago, but I wasn't actually a part of the C# 1.0 or 2.0 team.

When thinking about goals, I try to distinguish the "business" goals from the "technical" goals; these goals are strongly aligned, but they are different perspectives. From a business perspective, the primary goal of the C# product was – and continues to be – to create rich, compelling language tools that enable developers to take advantage of the .NET platform specifically, and more generally to improve the state of the art for the whole Windows ecosystem. Better tools means more productive developers, more productive developers means better applications for their users, and better applications means the platform becomes more attractive, and everyone wins.

From a language design perspective, there are a number of core principles that the design team goes back to over and over again. The language is intended to be a modern, relevant and pragmatic general-purpose programming language, used by professional programmers working in teams on line-of-business application software. It's intended to be object-oriented, but not dogmatically so; the language design team happily takes ideas from other programming paradigms and incorporates them where they make sense.

C# 1.0 started off as a pretty typical modern language. Obviously it was heavily influenced by C and C++; the design team sought to mitigate some of the shortcomings of C/C++ while still permitting access to the raw power of unsafe code. But in the decade since it has grown in leaps and bounds, adding a generic type system, sequence generators, functional closures, query comprehensions, dynamic language interoperability and lately, greatly improved support for writing asynchronous workflows. I am thrilled with how the language has evolved over the past twelve years, and it's been a privilege to have been a part of it for some of the most exciting changes. I'm looking forward to continuing to work in the C# ecosystem.

DNC: As a member of the C# team, what kind of back and forth did you have with Windows OS team? In fact for any language development, at what point is the OS something you start considering? Is that you can pretty much work in a silo and call people up when you hit a wall with respect to a feature, or is it a more collaborative endeavor?

EL: That has changed over time. In the distant past it was somewhat more "siloed"; for C# 5.0 though, the Windows team was very much involved. I *personally* had very little communication with the Windows team during the development of C# 5.0, but the C# program management team was in near-constant communication with their counterparts on the Windows team throughout the whole Windows RT effort. There were a number of technical issues that needed to be taken care of to ensure that there was as little "impedance mismatch" as possible between C# developers and the Windows RT programming model. In particular it was important that "async/await" met the needs of Windows RT developers using C#. My colleagues spent many long design sessions with the Windows team to make sure that the way Windows developers "do asynchrony" would make sense to C# developers and vice versa.

However this phenomenon is relatively recent. C# historically has not had that level of direct engagement with the Windows team because of course C# typically runs in the managed environment of the CLR, and C# programmers historically have typically used the BCL classes to execute operating system functionality. Since the C# team knew that the CLR and BCL teams would essentially be "brokers" for the operating system services, the C# team could concentrate more on designing a language that showcased the power of the CLR and the BCL, and let those teams manage the relationship with the operating system.

DNC: We were listening to your Podcast with the StackExchange team and you mentioned about the thing that's on top of your list of – "if I had a Genie to fix C#"... It was about unsafe array covariance. Could you explain the issue for our readers? What would the fix be like, in the language?

EL: Sure. First let's loosely define the somewhat highfalutin term "covariance". A proper definition would involve a quick course in category theory, but we don't have to go nearly that far to give the flavor of it. The idea of covariance is, as the name implies, that relationships "vary in the same direction" when some transformation is made.

In C# we have the rule that if T and U are reference types, and T is convertible to U by a reference conversion, then T[] is convertible to U[] by a reference conversion as well. This is said to be a "covariant" rule because you can transform the statement "T is convertible to U" into the statement "T[] is convertible to U[]" and *the truth of the statement is preserved*. This isn't the case in general; you can't conclude, for instance, that List<T> is convertible to List<U> just because T is convertible to U.

Unfortunately, array covariance weakens the type safety of the language. A language is said to be type safe when the compiler catches type errors – like assigning an int to a variable of type string, say – and prevents the program from compiling until all the type errors have been removed. Array covariance produces a situation where a type violation on an assignment cannot be caught until runtime:

```
static void M(Animal[] animals)
{
  animals[0] = new Turtle();
}
static void N(Giraffe[] giraffes)
{
  M(giraffes);
}
```

Because array conversions are covariant, the array of giraffes may be converted into an array of animals. And because a turtle is an animal, you can put a turtle into an array of animals. But that array is actually an array of giraffes. Therefore this type error becomes an exception at runtime.

This is egregious in two ways. First, it ought to be the case that assigning a value to a variable can always be type checked at compile time, but in this case it cannot. And second, this means that every time you assign a non-null reference to an array where the element type of the array is an unsealed reference type, the runtime has to interrogate the runtime type of the array and the runtime type of the reference assigned to verify that they are compatible. That check takes time! In order to make this bad feature work, *correct programs have to be slowed down on every such array access.*

> *If I had a genie that could fix everyone's code, I'd remove the unsafe covariant array conversion entirely...*

The C# team added typesafe covariance to C# 4.0. If you're interested in the design of that feature, I wrote a long series of articles during the design process; you can read them here: http://blogs.msdn.com/b/ericlippert/archive/tags/covariance+and+contravariance/

If I had a genie that could fix everyone's code, I'd remove the unsafe covariant array conversion entirely, and fix up everyone's code to use the typesafe covariance added to C# 4.0.

DNC: Before we get to your current job at Coverity, tell us a little more about what is Static Code Analysis?

EL: *Static analysis* is analysis of a program given only its source code. It's distinguished from *dynamic analysis*, which analyzes programs as they are actually running. Compilers do static analysis; profilers do dynamic analysis. Compilers use static analysis to do three things: first, to determine if the program is a legal program and give an appropriate error if it is not. Second, to translate legal programs into some other language – typically the language is some sort of bytecode or machine instructions, but it could be another high-level language. And third, to identify patterns that are legal but suspicious to produce warnings.

The kind of static analysis we do at Coverity is mostly of the third kind; we presume that the code is already legal code that will be checked by a compiler, but we do a far deeper static analysis of the code than most compilers do to find suspicious patterns and bring them to your attention. As is always the case, the earlier you find a bug, the cheaper it is to fix.

> *I spent about fifteen thousand hours carefully studying the design and implementation of C# compilers*

There are also other things you can do with static analysis; for instance, Coverity also makes a product which uses static analysis to try to find code changes that do not have an associated unit test. But the majority of what I do is in defect analysis.

DNC: How does your in-depth knowledge on C# add value to a company like Coverity?

EL: In two main ways. First, C# is a huge language now; the specification runs to about eight hundred pages. Line-of-business developers certainly do not have to know the whole language inside-out in order to use it effectively, but compiler writers certainly do! I spent about fifteen thousand hours carefully studying the design and implementation of C# compilers and learning from language designers like Anders Hejsberg, Neal Gafter and Erik Meijer, so I've got a pretty solid grasp on what makes a good C# static analyzer.

Second, I've seen thousands upon thousands of lines of buggy C# code. I know what sorts of errors C# developers make, which helps us figure out where to spend effort in static analysis. You want the biggest possible bang for the buck in static analysis; we don't want Coverity's automated checkers to find "theoretical" bugs or "style" bugs. We want to find *thank-goodness-we-found-that-before-a-customer-did bugs*.

DNC: Since starting with Coverity did you come across situations where you thought – "umm, it would help C# was more statically provable with this"?

EL: There are definitely features of C# that make it harder to do static analysis, but those are the same features that add great power to the language. Virtual methods, for example, enormously complicate static analysis because of course the whole point of virtual methods is that the actual method that executes is chosen based on the runtime type of the receiver.

As I said before, C# 1.0 was designed with the shortcomings of C/C++ in mind. The designers of C# 1.0 did a good job of that; it's a lot harder to write a buffer overflow, or a memory leak, or to use a local before it is initialized, or to accidentally use the same name to mean two completely different things. But what has been somewhat eye-opening to me since starting at Coverity is just how many of the defect patterns that the Coverity products check for in C and C++ apply equally well to modern languages like Java and C#.

DNC: We heard you developed one of the first 'Lord of the Rings' fan pages. Tell us more about how that came about and your interest in Books and Movies.

EL: My father read **The Hobbit** to me when I was very young, which started a lifelong enjoyment of J.R.R. Tolkien's writing; I've collected biographies of Tolkien as a hobby since I was a teenager. When I was studying mathematics at UW in the early 1990s the World Wide Web was a brand-new thing; one day I was "surfing the web" – a metaphor which makes not the slightest bit of sense, by the way – and I found an original-series "Star Trek" fan page. I thought that was an excellent idea, so I searched the entire internet – which did not take long in 1993 – and found all the FTP sites and newsgroups and whatnot about Tolkien, made a simple web page that was just a list of links, and put it up on the Computer Science Club's GOPHER server. As the web grew, more and more people linked to it and sent me the addresses of their pages. I kept adding more and more links to it, until eventually it just got to be too much; the web was too big. I stopped maintaining it and eventually my membership lapsed and it went away. You can still find it on the internet archive I'm sure, though it is not much to look at.

The thing is though, in that time companies like Open Text and Google started indexing the internet, and their search algorithms took into account things like how long a page had been up, how much it had changed over time, and how many inbound links there

were. Even after I stopped actively maintaining it, my little page scored high on all those metrics. As a result, for many years my name was the first one you'd get when you did a web search for "Tolkien". When the movies were announced a *lot* of people did exactly that. As a result I ended up being interviewed by several newspapers, I got email from one of Tolkien's grandchildren, the Jeopardy fact checkers called me up once to make sure that "what are the Ents?" was in fact the right question to whatever answer they had in mind, and so on. I had a lot of fun with it.

That said, I read very little "genre fiction" – science fiction, fantasy, mystery, and so on – these days. Most of my reading in my free time is non-fiction, popular history books.

I love movies and have friends over to watch movies frequently; our movie night choices are completely eclectic. We'll do recent Oscar winners one month and Hammer Horror the next.

QUICK BYTES

Born	November 30
Lives in	Seattle, Washington
Hobbies	Sailing small boats, playing the piano and editing books about C#
Current Work	Working on static analyzers for C# at Coverity

DNC: What's a typical day like for Eric Lippert when he is not programming?

It is **non-stop excitement**. My wife and I have a hundred-year-old house that we work at keeping from falling down around us. I have an International 14 – a tiny little overpowered sailboat – that I sail in the summer. I play the piano and the ukulele. I recently started teaching myself how to work metal, though I've put that on the back burner while I learn my way around the Coverity codebase. And of course I write and edit books about C#, write a blog about the design of C# (now at http://ericlippert.com), and answer user questions on StackOverflow. I like to keep busy.

DNC: Finally, to wrap up, Surprise our audiences with a hidden feature of C# not many know about.

C# is a big language and there are a lot of small features that people don't know about.

> *One of the ways that you know a language is getting really large, is when users submit feature requests for features you already have.*

Here are three little features that not a lot of people know about:

- You can put the prefix "global::" on a name to force the name resolution algorithm to start searching in the global namespace. This is useful if you are in the unfortunate position of having a name conflict between a global namespace and a local namespace or type. For example, if your code is in a badly-named namespace "Foo.Bar.System" then the name "System.String" will try to look up "String" in "Foo.Bar.System". The name "global::System.String" looks up the global namespace System.

- C# prohibits "falling through" from one switch section to another. What a lot of people don't know though is that you don't have to end a switch section with a break statement. You can force fall-through by ending a switch section with "goto case" and give it a case label. You can also end a switch section with a regular goto, return, throw, yield break, continue or even an infinite loop.

- One last clever little feature that few people know about: you can combine uses of the null coalescing operator to give you the first non-null element in a sequence of expressions. If you have variables x, y and z, all nullable ints, then the expression x??y??z??-1 results in the first non-null integer of x, y and z, or -1 if they are all null.

I often talk about odd features like this in my blog, so check it out if you want more examples. ■


Follow us on
Twitter
@dotnetcurry

SharePoint SERVER 2013 WORKFLOW

Pravinkumar Dabade shows us how to install and configure Workflow Manager for SharePoint Server 2013. We will see how to design, import, implement and deploy a workflow using Visio 2013 and SharePoint Designer 2013

Every iteration of SharePoint has made its Workflow capabilities richer and more powerful. For an Information Management system like SharePoint, data and document approvals are a common requirement. Today's business requirements are often more complex than a simple Approve/Reject and contain multiple steps involving multiple people. So demands from SharePoint's workflow system have increased manifold.

SharePoint 2013 has revamped its Workflow engine that makes business process management much easier. Before we get into the details, let's start with a quick look at what's new.

What's new in SharePoint Server 2013 Workflow

- SharePoint Server 2013 workflows are now using Windows

WORKFLOWS ON SHAREPOINT 2013 – YOUR TOOLBOX

Microsoft has provided us with a palette of tools for developing SharePoint Workflows. We can use any of the following Products:

01 SHAREPOINT DESIGNER 2013

a. SharePoint Designer provides a rich set of functionalities to design and develop the workflows for SharePoint Server 2010 as well as SharePoint Server 2013 platform.

b. SharePoint Designer is also considered the Primary Tool for designing Workflows.

02 MICROSOFT VISIO 2013

a. Visio 2013 is an Office application basically used for designing diagrams by using different Shapes and Connectors.

b. You can design workflows based on SharePoint 2013 workflow platform.

c. You can import the design of Visio into SharePoint Designer 2013 and vice versa.

d. The Visio integration is aimed at the Power Business Users responsible for designing Business Workflows.

03 INFOPATH 2013

a. You can use the InfoPath for designing the forms that can be used by the workflow users to input the data to the workflows.

b. For example – you can create an initiation forms, feedback forms and others.

c. Infopath's Workflow integration is for simple workflows like single step approvals etc.

04 VISUAL STUDIO 2012

a. Visual Studio 2012 is a product for developers used for developing different types of applications. We can use VS 2012 for designing, developing and deploying the Workflow for SharePoint Server 2013 as well.

b. Visual Studio is also used for designing custom actions for SharePoint which can be later used in SharePoint 2013 Designer tool in addition to designing Workflows for a specific task or a requirement.

Workflow Foundation 4.0. It uses the Workflow 4 activities to implement SharePoint Business Processes. Workflow 4 activity classes are implemented declaratively using XAML.

- In a SharePoint 2013 farm, Windows Workflow Foundation is hosted in Workflow Manager Client 1.0.

- The new Workflow Manager Client is the workflow execution engine for SharePoint 2013.

- SharePoint 2013 – provides the framework for SharePoint based workflows, which involve Documents, Lists, Users, tasks etc.

- From a backward compatibility point of view, 'SharePoint 2013 Workflow Interop' allows you to use SharePoint Server 2010 workflows (which are built on top of Windows Workflow Foundation 3.0) in SharePoint Server 2013 workflows (which are built on top of Windows Workflow Foundation 4.0).

Overall SharePoint Designer 2013 looks like a great tool for creating workflows and publishing them to SharePoint sites, let's dig deeper and see what it offers.

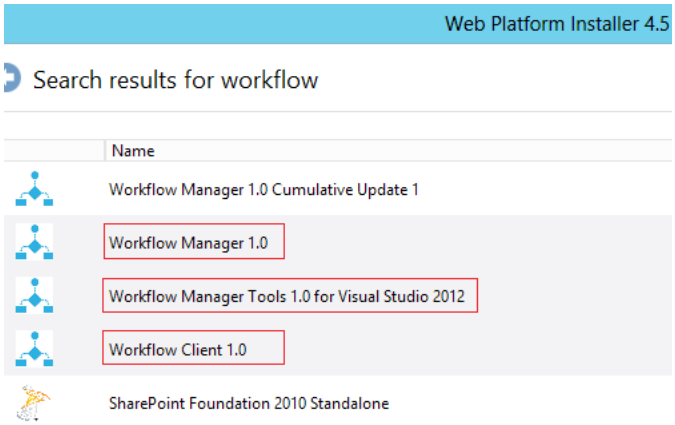
Setting up the Workflow Manager

The first thing we will have to do is to configure the Workflow Manager 1.0. So, let's open Web Platform Installer 4.5. In a search box, type workflow and press the Enter key. Now let's choose the following –

1. Workflow Manager 1.0 – is a Workflow host.

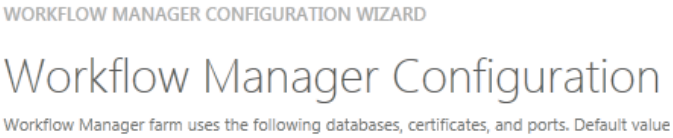
2. Workflow Manager Tools 1.0 for Visual Studio 2012 – for workflow development in Visual Studio 2012.

3. Windows Client 1.0 – Allows clients to communicate with the Workflow host.



Once all the above components have been installed, we'll configure the Workflow Manager. Start the Workflow Manager and click on "Creating a New Farm". It will start off the Configuration Wizard the first time around.

In this wizard, you will be asked to set the following -



1. SQL Server Instance Name.
2. Database Name.
3. Service Account.
4. Don't forget to select the option – Allow workflow management over HTTP on this computer.

Once the configuration is done, it will show you a summary.

Server Configuration

Now let's configure the workflow manager on the server to enable it to communicate over HTTP. Open SharePoint 2013 Management Shell and execute the following command –

```
Register-SPWorkflowService –SPSite "http://contoso/sites/msbi" –WorkflowHostUri "http://localhost:12291" –AllowOAuthHttp
```

(Replace Contoso and localhost with appropriate domain names)

DEVELOPING A WORKFLOW

We are all set from an infrastructure point of view, let's jot down the actual requirement or 'business case'.

The 'Business Case'

In our organization, when customer profiles are uploaded into Customer Profile Document Library, a manager will look at the customer profile and approve/reject the profile as per some pre-configured offline rules. So our Workflow has to route every Customer Profile to the manager for their approval.

Designing the workflow

For designing this workflow, we will use the following tools –
1) Microsoft Visio 2013.
2) SharePoint Designer 2013.

Getting Started with Visio 2013

In Visio 2013, when you create a new workflow instead of empty container, you get something called as the "Stage". Simple Workflows can have a single stage and complex workflows can have many stages. Inside the stage, you can organize the actions. All actions must be defined inside one of the stages.

When you create a SharePoint Workflow in Visio 2013, the default layout looks like below –

Visio provides us with a wide palette of 'Workflow Actions' out of the box. Some of the common ones are as follows –

1) **Action Shapes** – the action shapes are used for performing specific actions during the execution of the workflow. Few examples are like –

a. Add Comment – you can leave some comments in a workflow.
b. Assign a Task – assigns a workflow task to the user of a workflow.
c. Copy Document – copies document from the current list into a different document library.
d. Create List Item – creates a new list item in the list specified at design time.
e. Send an Email – when an event occurs in the workflow, you can send an email automatically with the predefined message to a particular user or group.

2) **Condition Shapes** – the condition shapes are used for applying conditions in the workflow.

a. If any value equals value – compares two values.
b. Person is a Valid SharePoint User – checks whether user is a registered user or group member on SharePoint site.

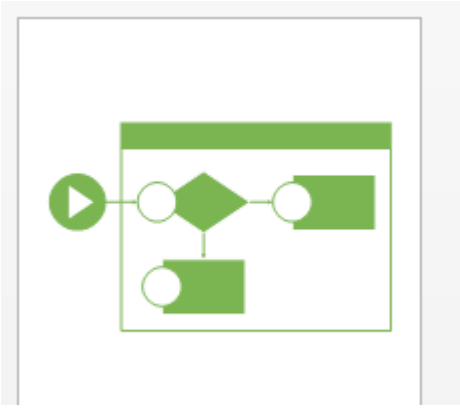
3) **Terminator Shapes** –

a. Start – begins the workflow. There can be only one start shape in a workflow diagram.
b. Stage – Contains any number of shapes and actions which must be defined inside the Stage.
c. Loop with condition – loops until a specific condition is met.

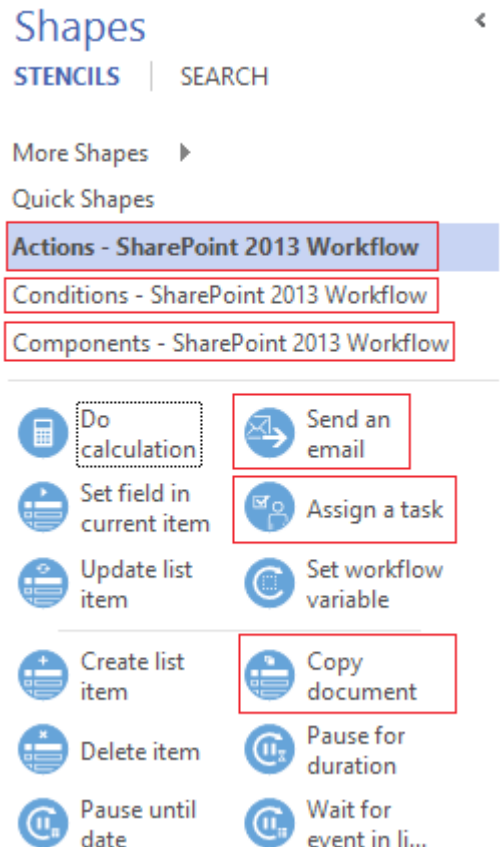
The Design

Now let's design a workflow diagram using Visio 2013 as per our scenario described above.

Start Visio 2013 and select the – New "Microsoft SharePoint 2013 Workflow" startup template.



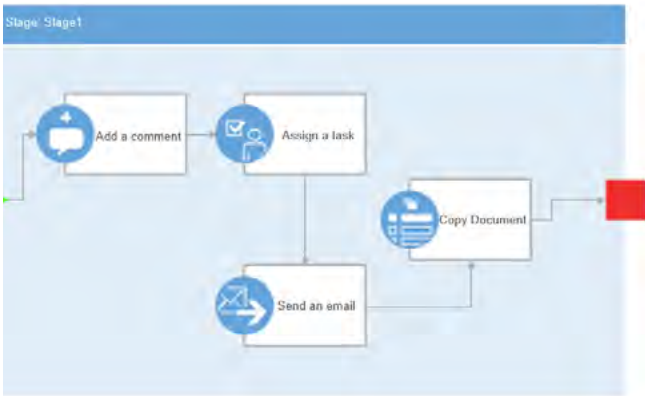
Once you click on template, pick "US Unit" (or the appropriate Unit for your country) and click on "Create" button. It will now show you a default SharePoint Workflow diagram with Start, Stage and End. On the left hand side it will show you a Shape task pane with all the available stencils as shown below –



Now let's add some actions in our stage 1 as described below –

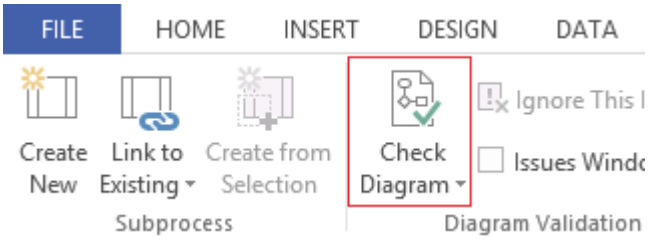
1) Add an action "Add a comment" in 'Stage 1'.
2) Add an action "Assign a task".
3) Send an email.
4) Copy document.

At the end, your workflow diagram should look like below –



Let's save the workflow diagram with the name "CustomerApprovalWF". After saving the workflow diagram, we can validate the workflow using the 'Check Diagram' option to run rudimentary checks for default values etc. in our workflow.

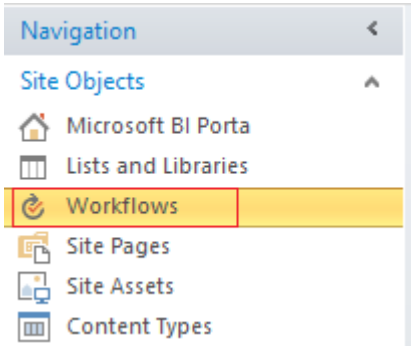
Click on "Process" tab and click on "Check Diagram" as shown below –



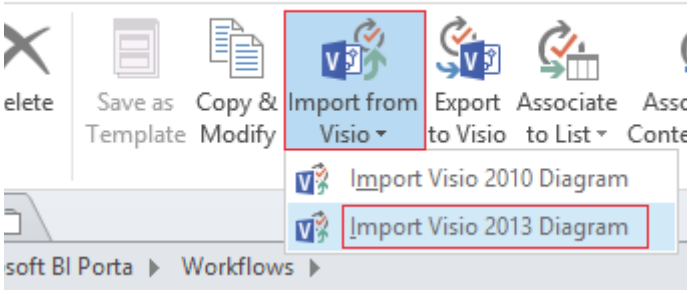
By this time, I assume, you have already created a SharePoint site, if not, go ahead and create one. Now it's time to create two document libraries, we'll give the following names for this example –

1) Customer Profiles.
2) Customer Profiles Status.

Now let's open SharePoint Designer 2013 and open the SharePoint site in which we have created the Document Libraries. On the left pane "Navigation", click on "Workflows" as shown here –



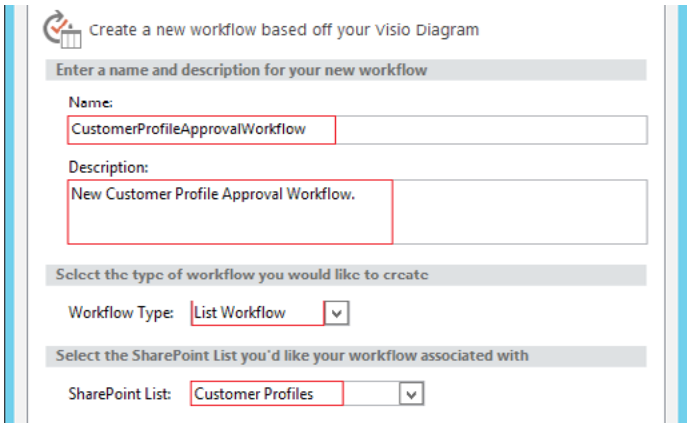
You will now see the default workflows associated with your site. Now from the "Workflows" ribbon, select "Import from Visio" and pick "Import Visio 2013 Diagram"



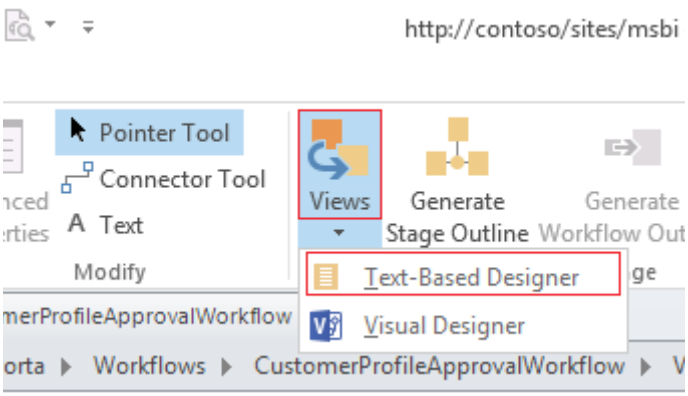
Select "CustomerApprovalWF.vxds" file and click on "Open" button. It will ask for the following –

1. The name of the Workflow.
2. Description of the Workflow.
3. Workflow type. [List or Site or Reusable workflow].
4. SharePoint list name with which you want to associate a workflow.

Let's write a name of the workflow as "CustomerProfileApprovalWorkflow". Write description as "New Customer Profile Approval Workflow". From the Workflow type dropdown list, choose "Reusable Workflow" and select the Document Library "CustomerProfiles" as shown below –

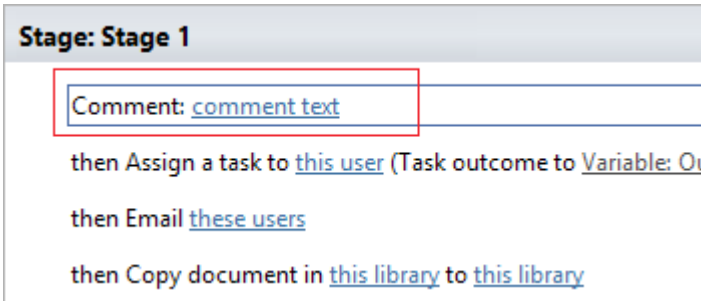


Click on the "OK" button to create the workflow. Once the workflow gets generated, it will show you the workflow in a Visual Designer inside the SharePoint Designer tool which allows you to modify the workflow using Visio shapes as per your process. In our case, we will not modify the workflow. We will now build the logic for this workflow using SharePoint Designer. To see the workflow in SharePoint Designer, click on the drop button "Views" from the "Workflow" tab and choose "Text-Based Designer" as shown below –



This will show you a workflow in SharePoint Designer in which now we can go and write the complete logic for Customer Profile approval.

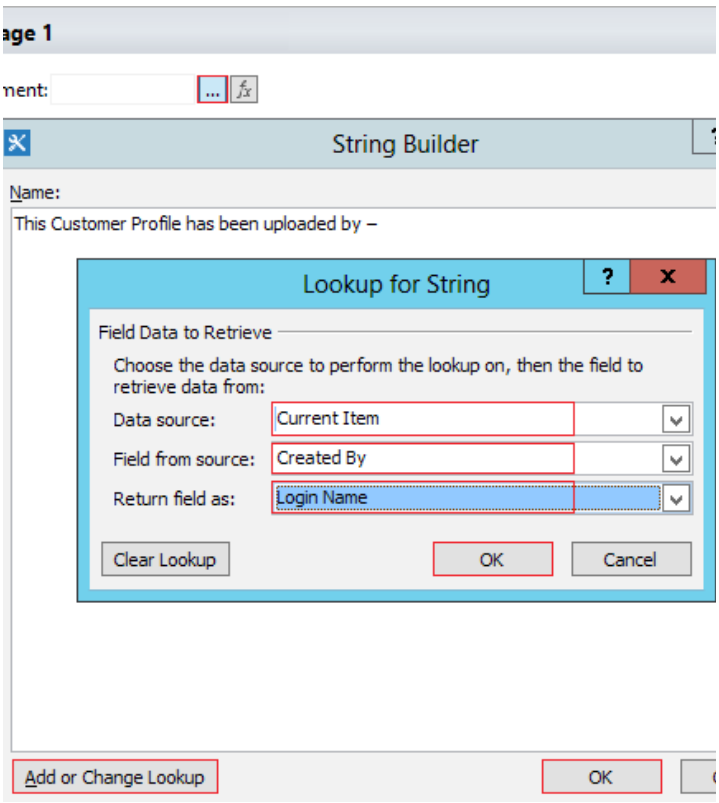
In Stage 1, the first option is to write a comment. Click on comment text link as shown below and write a comment as "This is a Customer Approval Workflow" –



Now it will show you the dialog box where we can write the text comments as well as we can embed the lookups in the message. Click on the button and write a comment as "This Customer Profile has been uploaded by – "and then click on "Add or Change the Lookup" button.

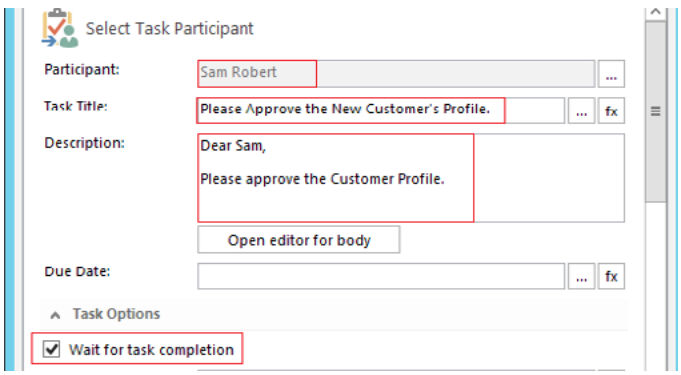
Note: You can setup your workflow to start off automatically on creating a List item as well (in this case it would be creation of Workflow as soon as the profile document is uploaded).

Select "Current Item" from the Data Source dropdown list. Make a choice of "Created By" from the Field Source and specify return field as "Login Name" as shown below –



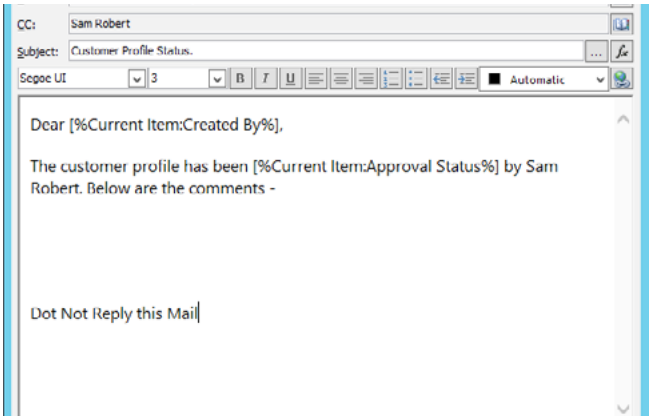
The second step is "Assigning a Task" to the user. In my SharePoint site, I have already configured one user who will act as a Manager for customer profile approval as a member of my site. I am going to assign a task to that user to approve/reject the customer profile. Click on "this user" link. It will now show you a "Assign a Task" window in which we will –

1. Select the participant.
2. Add the task title.
3. Write the description about the task.
4. If required, we can specify the due date.
5. From the task options section, check the checkbox "Wait for task completion".
6. Keep the default setting of Email options.
7. And choose "Default Outcome as Rejected" from the outcome options section as shown below –



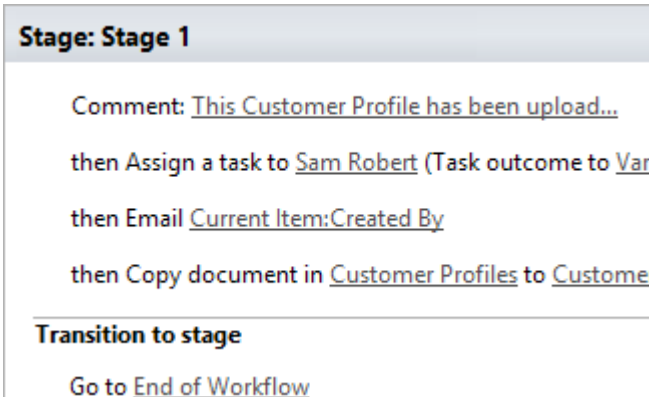
In the next step, we will assign 'send an email' to the creator of the document about the approval status of the workflow, with the comments from the approver. Click on "these users". Now format your mail as described below –

1. To – Current Item Created By (User who created current item).
2. CC – Sam Robert (Manager/Approver).
3. Add or Change Lookup with the message as shown below –



Once the email formatting is done, click on "OK" button.

Finally we will setup the Workflow Task to copy the document from "Customer Profile" into "Customer Profiles Status" document library with the status of the document as shown below –



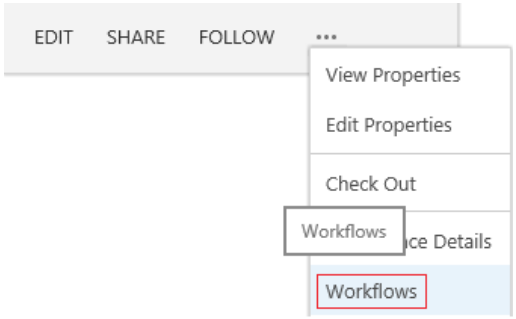
That is all. Your workflow is ready for the deployment.

Save the workflow and click on Publish workflow button from the "Workflow" ribbon.

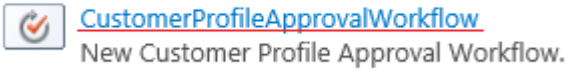
The Workflow in Action

Now let's create a word document and upload it into Customer Profiles document library of our SharePoint site.

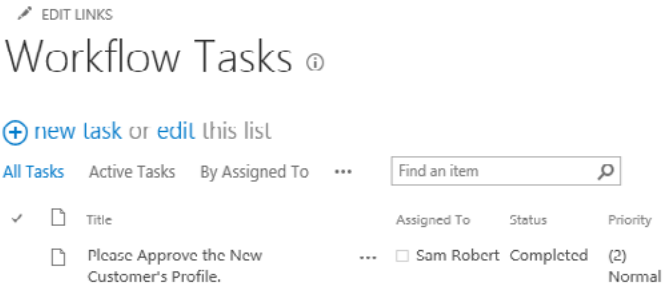
Once the document is uploaded, click on the "Workflows" by selecting the document as shown here –



You will see our workflow "CustomerProfileApprovalWorkflow". To start the workflow, click on the "CustomerProfileApprovalWorkflow". Now check your Workflow tasks list. You will see a task has been added in it. Complete the task by approving or rejecting the customer profile.



Check your email box and check the Customer Profile Status Library, you will see a mail and document copy.



CONCLUSION

To summarize, we saw how to install and configure Workflow Manager for SharePoint Server 2013 and use Visio and Sharepoint Designer to design, import, implement and deploy the workflow. We only had a very brief look at Workflow Development in SharePoint 2013. It has come a long way now and has enough components out of the box to support complex business processes. As with most Microsoft Business Software, it comes with excellent Tooling Support to make things as easy as possible for the 'non-coding' end user ■

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**

Responsive Image Viewer in ASP.NET MVC using **CSS 3 Media Queries**

ASP.NET Architecture MVP, ***Suprotim Agarwal*** walks us through the CSS – Media Queries Level 3 spec and shows us how to use them for building Responsive UI in ASP.NET MVC apps

HTML as we all know is the 'language' of the web and it gives 'structure' to the web pages that we see in our browsers. CSS on the other hand helps with the visual layout to the HTML markup. If we use a house as an analogy of a webpage; HTML is the pillar, the beam, the roof, the walls and the floor; while CSS is the tile used, the carpeting done (color, thickness, etc), paint on the inside and the outside and so on!

Essentially HTML is for providing structure and CSS is for managing content. Mixing the two up is like trying to build a house that uses colored drywalls (or colored plastering) and no paint work!!!

Just like in Architecture and Engineering, HTML and CSS are guided by established specifications. HTML 5 and CSS 3 therefore (loosely) imply the version of the specification. HTML v5 has been 'long time in the making' and has now reached 'Candidate Recommendation' stage or in software engineering terms, Release Candidate as of December 17, 2012.

CSS in the meanwhile, instead of releasing one monolithic spec, has broken things up into smaller 'Levels'. Today we will look at one such 'Level' - that's the 'Media Queries Level 3'. If you are getting the feeling this is all theory and no practice, hang in there, we'll build a nice showcase too. Before that, let's look into CSS and Media Queries in a little more detail.

CSS – MEDIA QUERIES LEVEL 3

CSS Media Types

CSS Media types have been around since CSS 2, however there was no built in support for 'querying' based on media type. One of the common examples of media type usage was to have different css for 'print' and 'screen' media types. One could define different style-sheets for these two media types as follows.

```
<link rel="stylesheet"
type="text/css" media="screen"
href="sans-serif.css">
<link rel="stylesheet"
type="text/css" media="print"
href="serif.css">
```

Apart from Media types, CSS expands Media support with addition of Media Features.

MEDIA FEATURES

Features help with the actual 'Media Queries'. The 11 features currently in the draft spec are

- width
- height
- device-width
- device-height
- orientation
- aspect-ratio
- device-aspect-ratio
- color
- color-index
- monochrome
- resolution
- scan
- grid

Let's see how Media Features are used in Media Queries to gain a better idea of why do we need Media Features in the first place!

CSS Media Queries

Level 3 draft proposal specifies 'Media Queries' as a syntax that contain a media type and one or more expressions that checks for conditions of a particular 'Media Feature'. For example

```
<link rel="stylesheet" media="print
and (monochrome)"
href="example-bw.css" />
```

The above has a 'media' query that uses the Media Features, screen as well as monochrome to imply that if the current output is going to the printer (as opposed to the screen etc.) and the printer is monochrome, one should use the classes in example-bw.css.

Media Queries and Responsive UI

As we saw above, we can pick and choose what CSS to apply based on media and condition of a media feature. If we go back to the list of features, we'll see that we have width/height, device-width/ device-height as features. These can be used in media queries to pick various CSS styles for various types of devices. Let's see a few examples

Example 1:

```
<link rel='stylesheet' media='screen
and (min-width: 701px) and (max-
width: 800px)' href='css/medium.
css' />
```

The above query implies that if the current media is a screen and its width is between 701 pixels and 800 pixels, it should use the medium.css.

Example 2:

The above query can be used in-line as well and would look something like this

```
@media only screen and (max-width:
850px) and (max-width: 850px)
{
 …
}
```

Media Features – width, device-width and what's the viewport meta tag?

When we setup a MVC application in Visual Studio, any template other than the 'basic' template includes the _Layout. cshtml file. In this file, in the <head> section, we'll see there is a line similar to the line below

```
<meta name="viewport"
content="width=device-width" />
```

ViewPort is actually CSS Level 2.1 defined spec, which was originally meant for defining the window used by continuous media like Video streams. However, with the advent of smart-phones, viewport is increasingly used to determine scaling and rendering in Mobile web browsers.

The above setting sets the 'width' media feature's value to the 'device-width' declared by the browser.

To Quote W3C specs – The *Width* Media Feature describes the width of the targeted display area of the output device.

Similarly *'device-width'* media feature describes the width of the device's entire screen irrespective of the browser window's width. On a desktop with screen resolution 1280x768 for example, the device-width is always 1280. On an iPhone -including iPhone 4- the device-width is 320.

We use the viewport meta-tag to set the width to the device-width like this:

```
<meta name="viewport" content="width=device-width" />
```

We are forcing the width of the browser to the device-width in the portrait mode. In landscape mode however, the width will change as per the browser's current width. We will see all of this in action.

With that, we wrap up our theory section, let's see how we can put what we learnt into action.

CREATING A RESPONSIVE IMAGE VIEWER USING CSS 3 AND MEDIA QUERIES ONLY

Today we'll build a very simple ASP.NET MVC application that will render the images posted by any user on Twitter. This of course requires us to log in to Twitter because twitter no longer supports un-authenticated queries. For authentication and querying we'll use LinqToTwitter, an excellent Twitter API library by Joe Mayo.

Getting Started

Step 1: We start off with a new ASP.NET MVC 4 Web Application (called Css3MediaBrowser here).

Step 2: In the next screen, we select the 'Empty' project template. This is because we want to see the CSS queries in action without the influence of any other styling or JavaScripts applied.

Step 3: Once the project is setup, we add reference to LinqToTwitter via the following command in the Nuget Package Manager Console

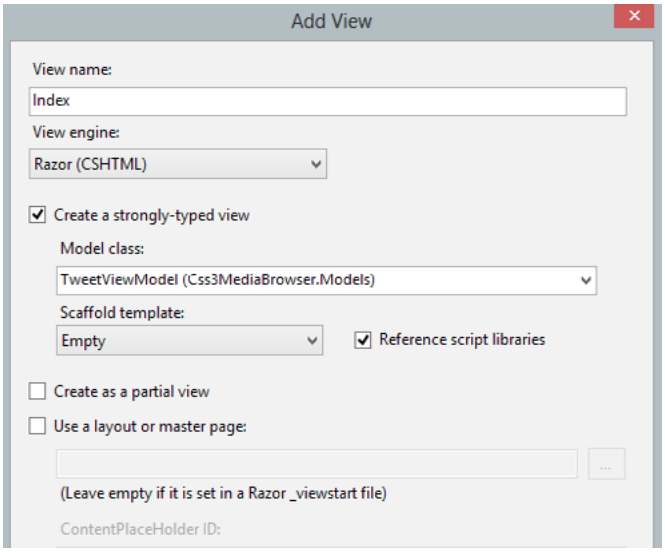
```
PM> install-package linqtotwitter
```

Setting up the View

Step 4: With LinqToTwitter in place, next we add our ViewModel class in the Models folder and call it TweetViewModel. It's a POCO with four properties. The ImageUrl contains the avatar of the user who tweeted, the ScreenName is the Twitter handle of the user who sent out the Tweet, MediaUrl is the Url of the image that we are going to use in the Image Browser and Tweet is the text of the Tweet.

```
public class TweetViewModel
{
 public string ImageUrl { get; set; }
 public string ScreenName { get; set; }
 public string MediaUrl { get; set; }
 public string Tweet { get; set; }
}
```

Step 5: Once the view model is in place, we build the solution. Next we add a folder Views\Home and initiate the Add View Wizard. As shown below, we create a Strongly typed view called 'Index' that uses the Razor syntax and uses our TweetViewModel class for data bind. We don't want any scaffolding code because we'll build our media browser from scratch.



Once we hit Add, Visual Studio generates the following boilerplate for us.

```
@model Css3MediaBrowser.Models.TweetViewModel
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
 <head>
  <title>Twitter Image Browser</title>
 </head>
 <body>
  <div>
</div>
 </body>
</html>
```

This is about as barebones as it gets!

Step 6: We first update the @model of the view and instead of getting just one instance of the TweetViewModel, we get an IEnumerable of the TweetViewModel

```
@model
IEnumerable<Css3MediaBrowser.Models.TweetViewModel>
```

Step 7: Our View (Index.cshtml) is going to consist of the following

- A label and Input for the screen name of the Twitter user

whose timeline we'll search for images
- A Submit button to post the screen name to the server
- A series of divs that will be laid out dynamically with the help of CSS.

The markup to achieve this is quite simple and shown below:

```
<body>
 <div>
  @using (Html.BeginForm("Index", "Home", FormMethod.
Post))
  {
   <h3>Get Recent Tweets by
<input id="screenName" type="text" name="screenName"   />
    <input id="submitButton" type="submit" value="Get" />
   </h3>
  }
  <div class="imageColumns">
   @foreach (var item in Model)
   {
    if (!string.IsNullOrEmpty(item.MediaUrl))
    {
     <div class="mediaDiv">
      <img src="@Url.Content(@item.MediaUrl)"
        class="mediaImage"
        alt="@item.Tweet" title="@item.Tweet"/>
     </div>
    }
   }
  </div>
 </div>
</body>
```

The Controller and Hooking up with Twitter

To keep the example simple, we will cut some corners here and instead of splitting our implementation up into the standard multiple layers, we'll build a 'fat controller' instead.

Step 8: Add a new Empty MVC Controller called HomeController

OAuth Recap

Before we get into the Controller implementation, a quick recap of how we'll use LinqToTwitter:

a. LinqToTwitter will require a ConsumerKey and ConsumerSecret for our application. We have to get this from http://dev.twitter.com . You can refer to our previous editions to see how we can create an application for ourselves on Twitter. Once you have the keys, add them to the appSetting section in the web.config as follows:

```
<add key="twitterConsumerKey" value="[the CONSUMER KEY
from Twitter]" />
<add key="twitterConsumerSecret" value= "[The CONSUMER
SECRET from Twitter]"/>
```

b. LinqToTwitter will check the SessionStateCredential store to see if it has a valid credential. If not, it will automatically redirect the user to Twitter's website for Authentication. Once the user has authenticated and provided permissions for our App to access their Twitter account, we'll get re-directed back to our application (all this is handled transparently by LinqToTwitter).

c. We have not built ANY backend to store Twitter's Authentication Token or the user's id, so every time our session expires, we'll have to Authenticate with Twitter again. This is just to keep things simple, in a real world application you should store the Authentication Token and reuse it till it gets revoked.

The Controller Implementation

The sole job for the Controller in our app is to return a list of TweetViewModel objects that is then used to render the View.

Step 9: We start the Controller implementation by declaring the following fields

```
private IOAuthCredentials credentials = new
SessionStateCredentials();
private MvcAuthorizer auth;
private TwitterContext twitterCtx;
```

All three types (or Interfaces) are defined in the LinqToTwitter namespace. The first one is a SessionStateCredentials() instance is the container for the Twitter provided authentication token. It's valid once user has approved our app to connect to Twitter.

The MvcAuthorizer takes care of the checking if the credentials from above are valid and if not, redirects to Twitter, intercepts the login success and stores the credentials in the Session.

TwitterContext is like a DB Context, as in it's the library used for querying Twitter just as if it were a database.

Step 10: With the field initialization done, we are ready to get data from Twitter and send it to the Index View. Core of the code is in the GetTweets method.

```
private ActionResult GetTweets(string screenName)
{
 if (credentials.ConsumerKey == null || credentials.
ConsumerSecret == null)
 {
```

```
  credentials.ConsumerKey = ConfigurationManager.
                      AppSettings["twitterConsumerKey"];
  credentials.ConsumerSecret = ConfigurationManager.
                    AppSettings["twitterConsumerSecret"];
 }
 auth = new MvcAuthorizer
 {
  Credentials = credentials
 };
 auth.CompleteAuthorization(Request.Url);
 if (!auth.IsAuthorized)
 {
  Uri specialUri = new Uri(Request.Url.ToString());
  return auth.BeginAuthorization(specialUri);
 }
 IEnumerable<TweetViewModel> friendTweets =
  new List<TweetViewModel>();
 if (string.IsNullOrEmpty(screenName))
 {
  return View(friendTweets);
 }
 twitterCtx = new TwitterContext(auth);
 IEnumerable<TweetViewModel> friendTweets =
 (from tweet in twitterCtx.Status
  where tweet.Type == StatusType.User &&
    tweet.ScreenName == screenName &&
    tweet.IncludeEntities == true
    select new TweetViewModel
    {
     ImageUrl = tweet.User.ProfileImageUrl,
     ScreenName = tweet.User.Identifier.ScreenName,
     MediaUrl = GetTweetMediaUrl(tweet),
     Tweet = tweet.Text
    })
    .ToList();
  return View(friendTweets);
}
```

We add the consumerKey and consumerSecret and add it to the instance of SessionStateCredentials(). Next we initialize the auth object and try to CompleteAuthorization. If authorization fails (session expired or first attempt), it reinitiates the authorization process by calling BeginAuthorization.

Once Authorization is done, we create an instance of TwitterContext(auth) and query it to get list of Tweets for the provided screenName. Our Linq query transforms the 'Status' object returned by the LINQ query into a TweetViewModel object. The GetTweetMediaUrl(...) extracts the URL of the Image if the Tweet has any images associated with it.

```
private string GetTweetMediaUrl(Status status)
{
 if (status.Entities != null &&
     status.Entities.MediaMentions.Count > 0)
 {
  return status.Entities.MediaMentions[0].MediaUrlHttps;
 }
 return "";
}
```

This wraps up the Controller and now we can finally go and checkout all the magic that CSS Media Queries can do.

Adding Content to the Structure

Earlier we setup the Index.cshtml, now that we have the Controller in place, we can build and run the application.

Basic Styling

By default, it looks up my 'timeline' for images I may have posted, and it shows one image per line, something like the following:

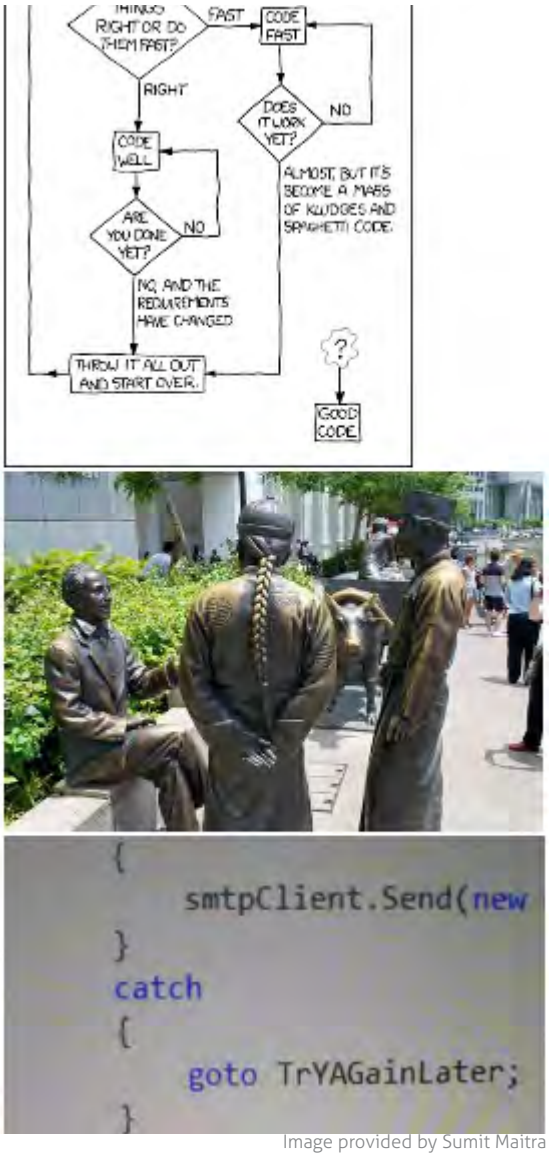


Image provided by Sumit Maitra

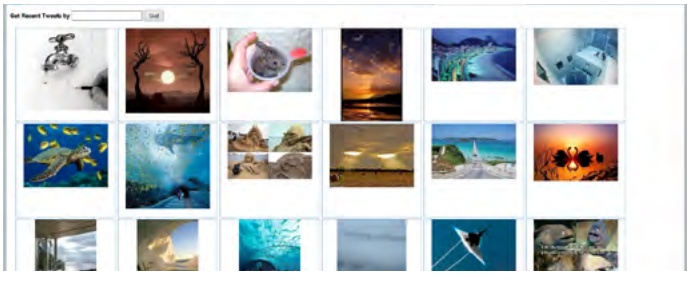
This as we can see is not pretty at all, so we go ahead and apply the following CSS styling

```
.imageColumns {
    margin: 10px;
}

.mediaImage {
    max-width: 85%;
    max-height: 200px;
}

.mediaDiv {
    float: left;
    width: 16%;
    margin: 2px;
    height: 200px;
    border: 1px solid #7ba9ca;
    display: table-cell;
    vertical-align: middle;
    text-align: center;
}

h3 {
    font-family: 'Segoe UI';
    font-size: 0.8em;
}
```

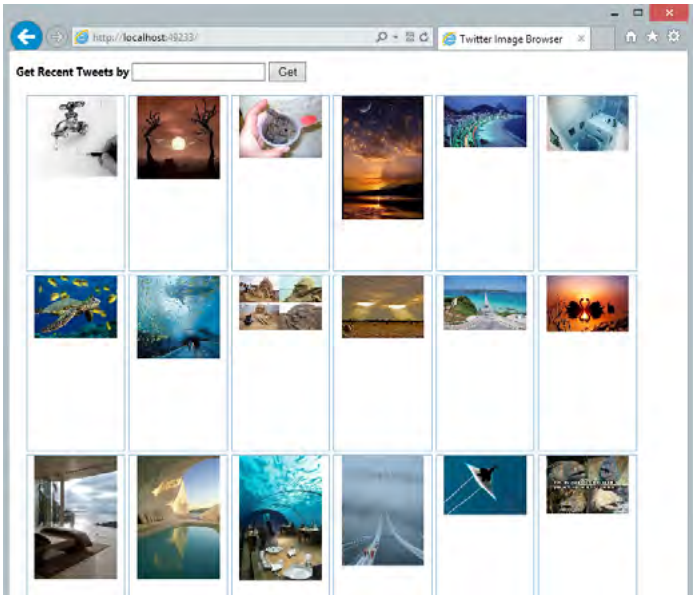
This sets up four CSS classes.

The 'mediaDiv' class arranges the image divs so that they float left instead of going one under the other. Once all horizontal spacing is occupied, it wraps the next div into the next line. They have been given a height of 200 pixels and a width of 16% which will roughly accommodate 6 images per line.

The 'mediaImage' class sets the max-width of the image to 85% so that there is some spacing between two horizontal images, and a max-height is set to the same as the height specified in the 'mediaDiv'. This ensures that the aspect ratio for the images are maintained when we change the browser width. Let's run this and see how it looks in various browser sizes. This time we 'Get' images for a Twitter handle called 'FascinatingPics'. As we can see below, we have a nice collage of the 20 latest images posted by this account.



But now let's reduce the size of the browser and see how this looks



As we can see, the images shrunk due to the width specified in % but the image divs are beginning to look odd with their fixed height.

Finally if we look at the results in a Mobile browser (simulated), it looks totally wacky.

So our one-size-fits all CSS style quickly breaks down as we view the page in different form-factors. We need to do something better.

Getting Media Queries into Play

As we can see from above, the Mobile view looks the most out of place. So let's fix that first using a Media Query to declare a new set of styles. But before we do that, let's do some house-cleaning first.

- We take our basic css from above (and probably defined in Index.cshtml) and put it in a style sheet in the Styles folder. We call it media-browser-hd.css.
- Next we add a reference to it in our cshtml's <head> section

```
<link rel='stylesheet' media='screen and (min-width:
1025px)' href='~/Styles/media-browser-hd.css' />
```

As we can see, the media attribute is providing a query that implies when the width of the browser is 1025 pixels or more, use the media-browser-hd.css style sheet.

So if we were to reduce the size of our browser to less than 1025 pixels, there would be no CSS applied again (because there are no other style sheets defined for other resolutions).

Styling for Mobile Devices

Well let's fix that for the Mobile browser first. We add another style sheet in the Styles folder and call it media-browser-mobile.css. We keep the names of the classes same but slightly tweak the width and height of the mediaDiv

```
.mediaDiv {
    float: left;
    width: 156px;
    margin: 1px;
    height: 100px;
    border: 1px solid #7ba9ca;
    display: table-cell;
    vertical-align: middle;
    text-align: center;
}
```

As we can see above, we've reduced the height to 100px and fixed the width to 156px. The exact number 156px comes from the math (320/2) - 2 x (1 + 1)!

Now what's that formula? Well, width of a 'popular smartphone' is declared as 320 pixels (portrait). We want two images to fit in those pixels. So we divide by 2 to get (320 /2 =) 160 pixels. We have a margin of 1px and a border of 1 px. Left and right border for two images = 2 x (1 + 1) = 4 pixels. 160 – 4 = 156px.

Okay, I went overboard there, but the fact you can target your CSS for well-known sizes, is what I am implying here. Now let's look at the changes in each style for mobile.

```
.imageColumns {
    margin: 0px;
}
```

The imageColumns no longer has a margin as its set to 0 as opposed to 10 for the desktop style sheet.

```
.mediaImage {
    max-width: 85%;
    max-height: 100px;
}
```

For the Image itself, we've reduced the max-height to match the height from the mediaDiv style.

```
h3 {
    font-family: 'Segoe UI';
    font-size: 0.6em;
}
```

Finally the heading size is reduced from 0.8em to 0.6em to reduce the text size for the input box and button on top of the page. The CSS is now all setup. We hook it up in the Index.cshtml as follows:

```
<link rel='stylesheet' media='screen and (max-width:
480px)' href='~/Styles/media-browser-mobile.css' />
```

Let's refresh our mobile browser and check

Sweet! As we can see the image above, width is working and the browser is adapting itself beautifully.

Handling a third form factor and embedding media queries in existing CSS

So far we've seen two form-factors. But remember between 481 pixels and 1024 pixels of screen-width, we have no CSS styling applied. So let's fix that.

This screen size is now very common with Tablets and handheld

computers, so we'll call this the Tablet settings. But instead of creating a third settings file, we'll simply put the styles in the media-browser-hd.css file. You may choose to do this if there are common styles between the two form-factors, or you can go the separate file per form-factor route. I am doing it for academic reasons here.

Open the media-browser-hd.css and put in the following

```
@media only screen and (max-width: 1024px) {
}
```

This implies that the enclosing styles will be applicable to browsers up to the width of 1024 pixels.

We adjust our style accordingly and the final set of styles are as follows:

```
.imageColumns {
 margin: 5px;
}
```

Set Margin to 5px instead of 10 in the desktop style.

```
.mediaImage {
 max-width: 85%;
 max-height: 150px;
}
```

Set max-height to 150 to match the height of the mediaDiv

```
.mediaDiv {
    float: left;
    width: 24%;
    margin: 1px;
    height: 150px;
    border: 1px solid #7ba9ca;
    display: table-cell;
    vertical-align: middle;
    text-align: center;
}
```

Set the Width to 24% giving us about 4 images per line and the height to 150px so that landscaped images don't look too odd.

```
h3 {
    font-family: 'Segoe UI';
    font-size: 0.7em;
}
```

Finally we update the font-site to be 0.7em

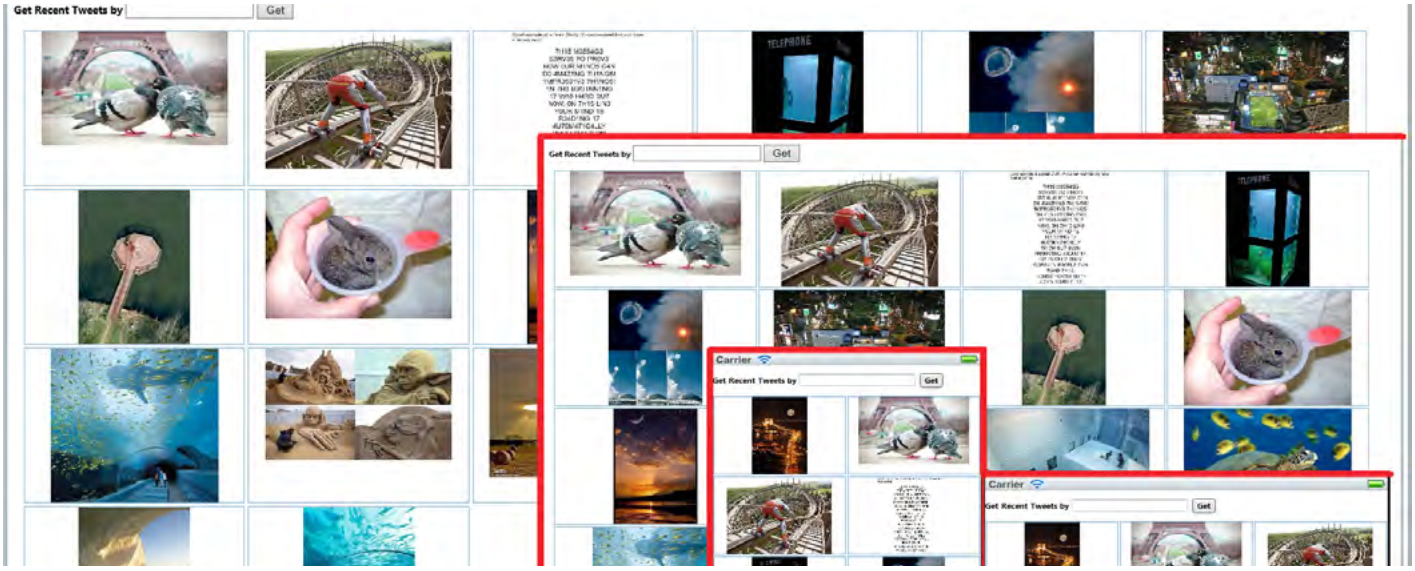
Updating the Media query for the media-browser-hd.css

Now that we've two types of styles defined in one CSS file, we have to ensure that the file is actually included for both types. We revisit the Index.cshtml and update the condition to include media-browser-hd.css as follows

```
<link rel='stylesheet' media='only screen and (min-width:
481px)' href='~/Styles/media-browser-hd.css' />
```

Essentially instead of declaring a minimum width of 1025px, we've reduced it to 481px so that the desktop style sheet (*-hd.css) is included for any screen 481 pixels and above. Thanks to the inner media query that has a max-width set to 1024, screen width between 481 and 1024 are handled by the tablet styles, inside the inner query.

We are all set, let's run the application and search for images posted by the Twitter user 'fascinatingpics'. I have composited the images from each form factor but we can see our media queries are in action and each form-factor shows a layout that looks nice and appropriate for the screen size.



Things to note – Caveats, ifs and buts

Well, that neatly sums up this example where we highlighted one way to leverage capabilities of CSS media queries. We only used the device width feature, but there are 11 other features for us to explore if we need them. Idea was to introduce you to them so that you can utilize them when the need arises.

Key takeaways

1. We have NO JavaScript references in our project whatsoever.

2. CSS Media Queries are well supported in IE9+, Firefox 3.5+ and most versions of Chrome. This app was built using IE10 and verified on Firefox 20.0.1 on Windows 8.

3. We did not investigate any techniques to adapt to different image sizes based on form factor so this implementation has a disadvantage where it's pulling the same amount of data on all screens which might be a problem for high latency networks like some mobile broadband. This can however be mitigated with some help from the Image source. Twitter supports multiple image sizes and we could use that to fetch different 'versions' of the same image.

4. Older browser support: This is a thorny issue and often we simply cannot leave behind folks that are behind on the adoption curve. In such cases, there are JavaScript shims available. One of the popular ones is Respond.js. You can check it out at http://wordpress.org/extend/plugins/respondjs/

5. The code for Twitter Authentication is not vetted from a security air-tightness point of view. So for a more 'secure' way, use ASP.NET's Twitter Authentication using DotNetOpenAuth and then use LinqToTwitter to query if you want.

CONCLUSION

Responsive UI design for the web has got a huge boost with CSS Media Queries Level 3. This article is aimed to serve as a primer to Media queries. Feel free to explore various features that are available and adapt them to serve up the best possible Views for the maximum percentage of your users ■

The source code can be downloaded from our GitHub repository at bit.ly/dncm6-cssmq

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

Extending Team Foundation Server 2012

ALM MVP *Subodh Sohoni* shows us the various extensibility points available in TFS and how to leverage them to align TFS to (existing) processes instead of the other way round!

Services of Team Foundation Server 2012 (TFS 2012) have a rich set of features to support generic application lifecycle management out of box. These services are to provide support of version control, build management and project management through work item tracking. In addition to these core services, TFS 2012 also offers reporting of data generated during development activities and a collaboration portal for each software project undertaken by the team. It supports the process models based upon CMMI, Agile and Scrum.

All these features are provided with certain default settings. These settings of the features make TFS behave in a certain way. It is the way these features are commonly used. But, it is also true that the way each feature is designed, cannot satisfy everyone's requirement. In fact, more the generic nature of these features, more are the chances that it may not satisfy anyone. The way organizations behave are, as per the set processes that they have implemented over the years and those processes may not be supported by the default behaviour of TFS. It is possible that those processes have evolved over the lifetime of the organization and embodies the decisions made during each unique situation that the organizations passed through. It is very difficult to say if processes followed by an organization are wrong, although those may not be exactly as implemented in TFS. Processes often gives a unique identity to the organization and those should not be changed if those processes are working for that organization. Tools like TFS may provide efficient ways to implement the same processes and flexibility of the tool to accommodate various processes of the organization, becomes its strength. TFS 2012 does provide many different opportunities to us to extend its services, so that we may implement specific and not just standard processes with it.

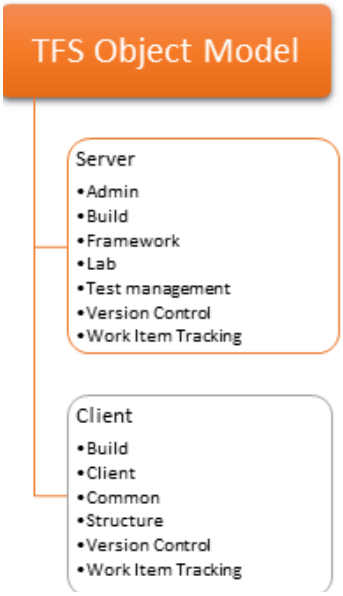
Start of the implementation of a process is with the Process Template which is a set of configurations that are applied to a team project when it is created. It provides configuration of TFS 2012 that has pre-built process templates and supports implementation of CMMI, Agile and Scrum. Organizations can pick any one of them as a starting point but need not be satisfied with settings that are out of box.

The strongest point of these process templates is that they are customizable. It is possible to:

- Create new work item types. For example Epic work item type as part of Agile process
- Groups of team members who need to be given some unique set of permissions
- Pre-populate iterations and areas as a common structure used in the organization
- Create and populate customized reports

Many more such customizations are possible. A Team Foundation Server Power Tool (http://www.microsoft.com/en-in/download/details.aspx?id=35775) makes it easy to customize processes as you need.

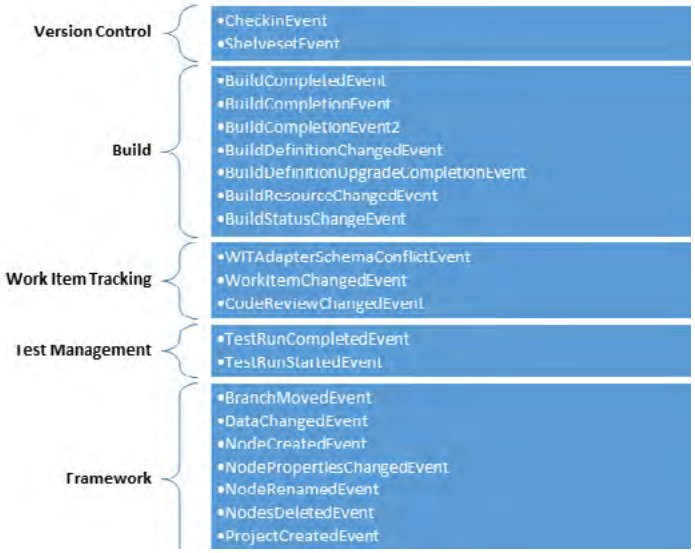
TFS 2012 is a set of WCF services. These services are exposed for extension by a well published object model. This object model is encapsulated in components i.e. TFS APIs. The structure of these APIs is:



One of the most popular use of this object model is to write an event handler that executes when a certain event is raised by TFS. For example, when someone creates a new work item, we may want to check if the work item type of that newly created work item is a 'User Story' and spawns a set of work items of the type 'Task' to implement that user story. In this case, we can create an event handler for the event of 'WorkItemChanged'. For this, we can write a class that implements Microsoft.TeamFoundation.Framework.Server.ISubscriber interface. In the ProcessEvent method, we can write the code that will execute when the event happens. Snippet of boilerplate code is as follows:

```
public EventNotificationStatus ProcessEvent(TeamFound
ationRequestContext requestContext, NotificationType
notificationType, object notificationEventArgs, out int
statusCode, out string statusMessage, out Microsoft.
TeamFoundation.Common.ExceptionPropertyCollection prop-
erties)
{
    statusCode = 0;
    statusMessage = string.Empty;
    properties = null;
    if (notificationType == NotificationType.
     Notification && notificationEventArgs is
WorkItemChangedEvent)
{
    string machineName = System.Environment.MachineName;
    TfsTeamProjectCollection tpc = new
         TfsTeamProjectCollection(new Uri("http://" +
         machineName + ":8080/tfs/DefaultCollection"));
    WorkItemChangedEvent ev = notificationEventArgs as
          WorkItemChangedEvent;
    WorkItemStore ws = tpc.GetService<WorkItemStore>();
    Microsoft.TeamFoundation.WorkItemTracking.
         Client.WorkItem wi = ws.GetWorkItem(ev.
               CoreFields.IntegerFields[0].NewValue);
    Project project = ws.Projects[ev.PortfolioProject];
    if (wi.Type ==  project.WorkItemTypes["User Story"]
         && wi.RelatedLinkCount == 0)
         {
            Microsoft.TeamFoundation.WorkItemTracking.
              Client.WorkItem NewTask = new Microsoft.
              TeamFoundation.WorkItemTracking.Client.
            WorkItem(project.WorkItemTypes["Task"]);
            NewTask.Title = "Create architecture diagrams
                  for user story " + wi.Id;
            WorkItemLinkTypeEnd linkTypeEnd =
                  ws.WorkItemLinkTypes.
                  LinkTypeEnds["Parent"];
            NewTask.Links.Add(new
                  RelatedLink(linkTypeEnd,wi.Id));
            NewTask.Save();
         }
      }
     return EventNotificationStatus.ActionApproved;
}
```

Similar to 'WorkItemChanged' event, TFS raises following events for which we can write various event handlers:



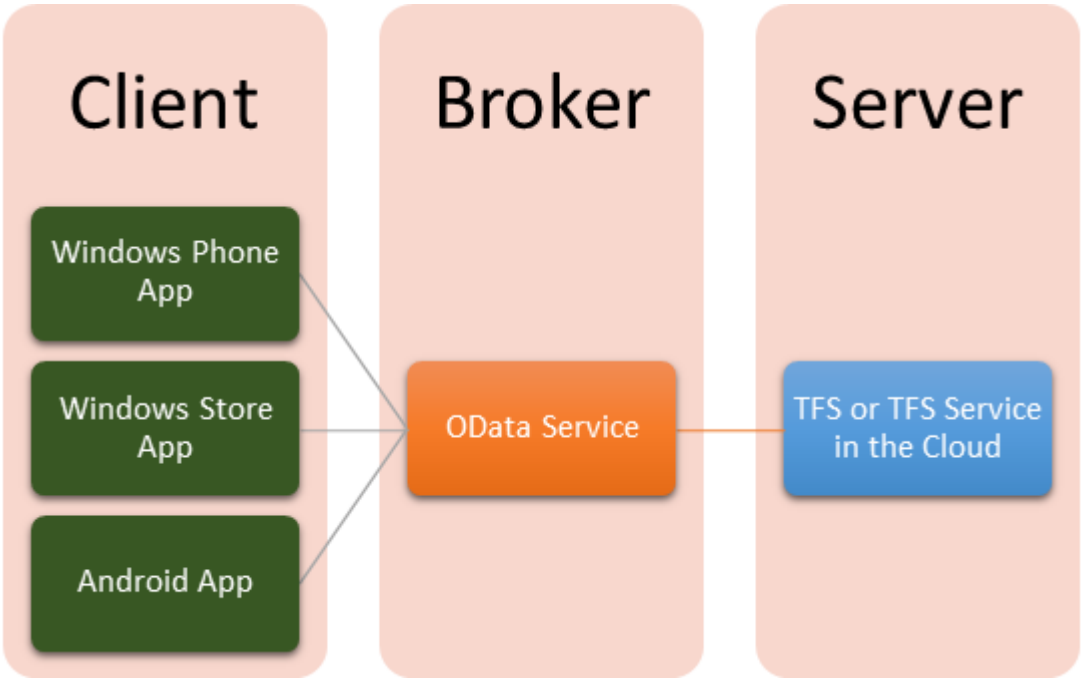
Since these event handlers execute on TFS itself (in fact, within the process of TFS), these can be written using any of the Microsoft .NET languages. It does not matter which client performed the activity that triggered the event, which means that even if your project is a Java project, the event handler will still have to be written in .NET.

The same Team Foundation Object Model can also be used to access Team Foundation Services from standalone applications. For example, we can create new Requirement type work items when a requirement document is approved in SharePoint. For this purpose, we can write a SharePoint workflow application in which we can refer to the TFS APIs, use those to create new Requirement type work items as desired and then associate that workflow with the appropriate library on the SharePoint portal. Following code shows how to do that (Some of the methods to handle worksheet data are not shown as they are secondary to the purpose)

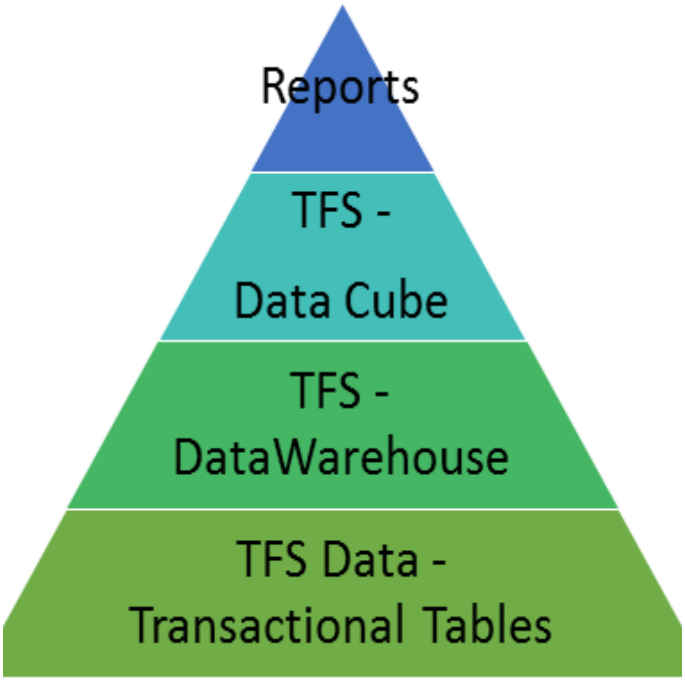
```
private void CreateRequirements()
{
    prj = (string)workflowProperties.Item["Team Project
                                                Name"];
   collection = new TfsTeamProjectCollection(new
          Uri("http://ssgs-vs-tfs2012:8080/tfs/
           DefaultCollection"));
   store = collection.GetService<WorkItemStore>();
   SPWeb web = workflowProperties.Web;
   SPFile file = web.GetFile(@"http://ssgs-vs-tfs2012/
   sites/DefaultCollection/SSGS%20CMMI/Requirements%20
   Documents/RequirementDocument.xlsx");
    SPFileStream dataStream = (SPFileStream)file.
    OpenBinaryStream();
    SpreadsheetDocument document = SpreadsheetDocument.
    Open(dataStream, false);
    WorkbookPart workbookPart = document.WorkbookPart;
    IEnumerable<Sheet> sheets = document.WorkbookPart.
       Workbook.GetFirstChild<Sheets>().
       Elements<Sheet>();
    string relationshipId = sheets.First().Id.Value;
    WorksheetPart worksheetPart =
      (WorksheetPart)document.WorkbookPart.
                       GetPartById(relationshipId);
    Worksheet workSheet = worksheetPart.Worksheet;
    SheetData sheetData = workSheet.
                         GetFirstChild<SheetData>();
    IEnumerable<Row> rows = sheetData.
                            Descendants<Row>();
    int i = rows.Count();
    foreach (Cell cell in rows.ElementAt(0))
    {
        string Title = GetCellValue(document, cell);
        WorkItem Requirement = new WorkItem(store.
           Projects[prj].WorkItemTypes["Requirement"]);
        Requirement.Title = Title;
        Requirement.Save();
    }
}
public static string GetCellValue(SpreadsheetDocument
document, Cell cell)
{
    SharedStringTablePart stringTablePart = document.
         WorkbookPart.SharedStringTablePart;
    string value = cell.CellValue.InnerXml;
    if (cell.DataType != null && cell.DataType.Value ==
     CellValues.SharedString)
    {
        return stringTablePart.SharedStringTable.
           ChildElements[Int32.Parse(value)].InnerText;
    }
    else
    {
        return value;
    }
}
```

Another use of this object model is to create custom check-in policies. Such a custom check-in policy will exclusively use client side objects of Team Foundation Server. Checks on the code or associated work items can be done by using those. For example, we can check if the code being checked in has an associated work item of a specific type like a 'Code Review' and make further check on it to ensure that it is having the state as 'Approved', for allowing the check-in to succeed. Since check-in policy is enabled on TFS but executes on the client side, it has to be developed using a language that is appropriate for the client environment. To develop check-in policies for Visual Studio, we can use Microsoft .NET languages, but for Team Explorer Everywhere 2012, that is a plug-in to Eclipse, we can use TFS SDK 2012 (For Java). It can be downloaded from http://www.microsoft.com/en-us/download/details.aspx?id=22616.

Many of the applications that use the Team Foundation Object Model are rich client applications so that they can reference the assemblies that encapsulate these objects. Size of such applications along with the TFS API assemblies is not optimized for lean environments like Windows Phone, Android Apps and Windows Store Apps (for Surface). For such environments, we need a broker that will consume TFS data using heavy TFS API assemblies and then in turn publish that same data as a lean HTTP service.



A sample implementation of TFS OData Service and its clients can be downloaded from http://www.microsoft.com/en-in/download/details.aspx?id=36230. OData service can be published on a local IIS server to make on-premise TFS accessible to tablets and also been published on Microsoft Azure to make it accessible from anywhere in the Internet. TFS Service (in the cloud https://tfs.visualstudio.com) also supports access through OData service.



TFS 2012 provides an infrastructure for publishing reports related to status and health of the development projects. It stores the data of each artefact created during the development process. Those artefacts can be the code, work items, builds, team members, teams, groups, iterations and many more. Whenever any event occurs like check-in, creation of work item, completion of build etc. at that time data is generated by TFS. This data is stored in the transaction data tables that are highly normalized, hence not suitable to be used for reporting. That data is then processed in the non-normalized form that is called TFS Datawarehouse and then aggregated in a data cube. Whenever the definition of work items is customized, the structure of the datawarehouse and the cube is also updated as appropriate. Data Cube or DataWarehouse forms the basis for reports. Although various process templates define appropriate reports out of box, those are just the beginning of reporting. We can create as many custom reports as the organization needs for management of projects using the same data cubes and warehouse. These reports can be ad-hoc reports created as Excel Pivot Tables or Pivot Charts or they can be predefined reports to be hosted in SQL Server Reporting Services or any other reporting client that can use SQL Server Analysis Services.

This multitude of options to extend TFS makes it one of the most flexible set of services for Application Lifecycle Management. While being flexible, it also is the most powerful and integrated solution for ALM ■

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

Head First Into ASP.NET Web API - Media Formatters

Suprotim Agarwal introduces ASP.NET Web API briefly before walking us through what are Media Formatters in Web API and how to build one for ourselves

> ASP.NET Web API is a framework that makes it easy to build HTTP services on top of the .NET framework

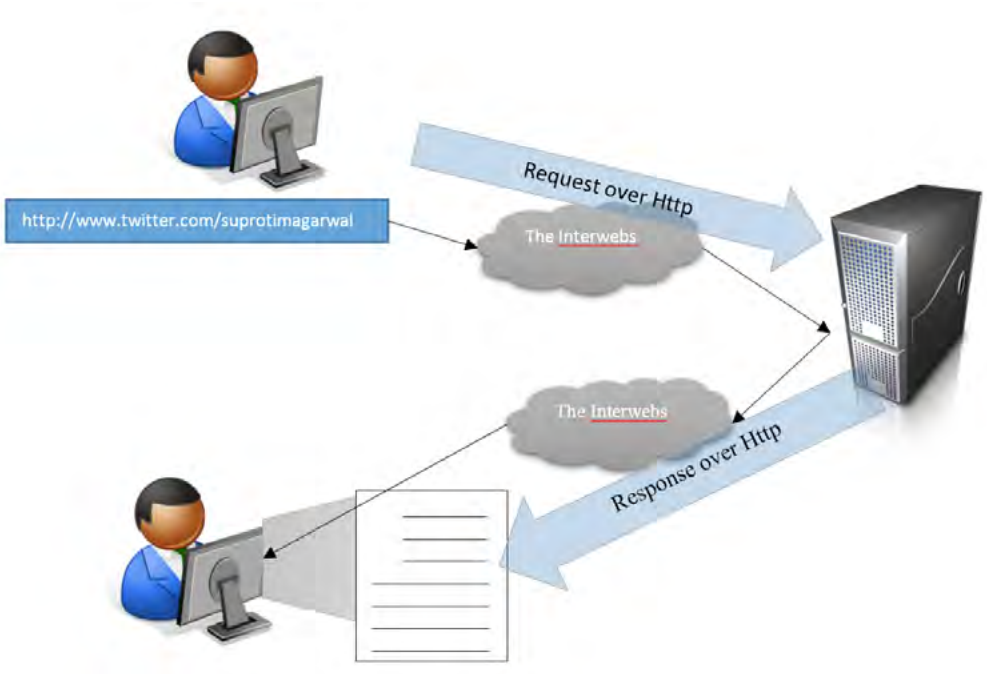
Microsoft introduced ASP.NET Web API as a lightweight HTTP based web services framework with the launch of Visual Studio 2012 and .NET Framework 4.5.

Prior to Web API, communicating over HTTP in Microsoft's dev stack was a colorful mess comprising of WS-* stack. There were other third party libraries that were built ground up on .NET, like ServiceStack.

With Web API, Microsoft took a clean break and built a framework that stuck to the basics of HTTP without introducing any complicated stubbing or end point proxies.

After a long period of incubation in the WCF team, the Web API team was merged with the ASP.NET MVC team. A lot of the final Web API's end point implementations thus mimic MVC.

If all this HTTP service talk sounds confusing, let's see if this diagram helps.



• In the diagram we just saw, a user is sitting at the computer and types in a URL in the browser. Notice that apart from the URL he also types in /suprotimagarwal which tells the server 'something'.

• The browser sends this request over HTTP to (GET) the actual site. The discovery process of www.twitter.com leverages the routing mechanism of the internet and it happens over port 80 using HTTP.

• The request is eventually sent to the Twitter server where it is processed by a web application running a service on top of HTTP at port 80.

• This service then interprets 'suprotimagarwal' in the URL as a request to GET (as in the HTTP request keyword) the profile of suprotimagarwal. Thus it makes appropriate backend calls to retrieve the user profile data and sends it back to the requestor. In this case the user's browser renders the data that came in. Note, sometimes, the raw data may not be sent back, instead the data is used to build the respective view and the complete view (in html) may be returned, but in context of Web API, it's not a view, only the data. Keep the term 'Content Type Negotiation' in mind, we will circle back to it in the next section.

Overall the idea of Web API is to offer services that work seamlessly over HTTP as we saw above. This is a great way to build public APIs for multiple client platforms.

HTTP AND CONTENT TYPES

Whenever we think of HTTP URLs and web browsers, we picture HTML being received by the browser and then rendered. However, there are instances when our browser renders an image by itself, or plays music or shows a PDF document in the browser itself. These are instances where data/content other than html was sent over and the browser 'understood' the 'Content Type' and acted appropriately. For example, if we have link to a PDF and the browser has a plugin for it, it can directly display the PDF (in browser).

When you build Web API based services, you can return data as JSON by default. However you can send out other types of content as well. For example, if you are sending out Calendar information, you can send it out as an iCal, when sending out graphs data you can send out the rendered image and so on. What type of data is acceptable to the client and what can the server provide is a 'negotiation' that happens between the client and the server before either a content type is agreed upon or a 406 (Not Acceptable) error is sent by the server implying, it cannot provide data in any of the requested Content Types.

MEDIA FORMATTERS AND CONTENT TYPES

As I mentioned briefly above, Web API services return data in JSON or XML. This is because the data returned by the Controller is converted into JSON or XML by a component referred to as Media Formatter. Web API has two of these out of the box, no points for guessing they are for JSON and XML formats.

Media Formatters are the extension points that allow us to plug in our own Formatter and thus help our Service support more Content Types.

DEFINING THE PREMISE

Today we will see how we can build a Web Service that returns Appointments in a rudimentary Timekeeping application that maintains 'Timecards'.

On top of the Service, we will have a Single Page application that uses Knockout JS to render basic CRUD screens for Timecards. We will also create a Media Formatter that formats these 'Timecards' into the VCalendar format so that they can be downloaded and imported into our Calendaring tools like Outlook etc.

BUILDING A 'TIME CARD' SERVICE

We will start off by creating a new MVC 4 project in Visual Studio and call it TimeKeepr. From the templates list, we select the 'Web API' template to get started.

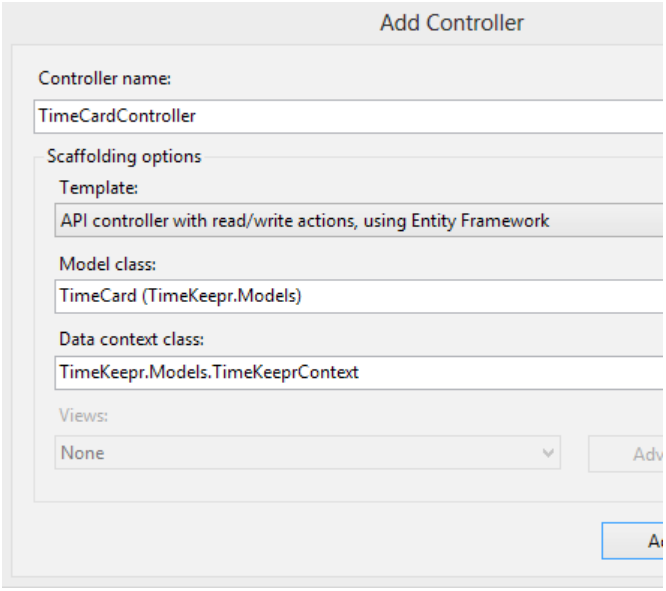
First up, let's build the model and the controller.

THE MODEL AND (WEB) API CONTROLLER

Our Model is very simple. It has the following properties. As we can see, it's a bare-bones definition of a Task, it simply maintains Summary, Description, StartDate and EndDate.

```
public class TimeCard
{
 public int Id { get; set; }
 public string Summary { get; set; }
 public string Description { get; set; }
 public DateTime StartDate { get; set; }
 public DateTime EndDate { get; set; }
 public string Status { get; set; }
}
```

Once the model is added, we build the app, and then add a TimeCardController with the following settings. As you can see, we are creating API controllers as opposed to MVC Controllers so no views are going to be generated.



The DBContext that is created by the scaffold pretty much takes care of the database persistence. Today we will cut some 'pattern' corners and leave the DBContext instantiation in the controller itself. For a production system, it should be a part of the Repository and we should be injecting repository instances into the controller.

THE VIEW USING KNOCKOUT JS

We will build a single page View with Add and Edit functionality. The view will use the Knockout JS library to implement two way data-binding. This will give us a nice and responsive UI. We will post data back to the server using Ajax. We will also use a plugin by Ryan Niemeyer that will help us control Knockout's two way data-binding by giving us options to either commit or roll-back any changes.

SETTING UP DEPENDENCIES

We'll update our jQuery and KO dependencies from the Nuget Package Manager using the following commands

```
PM> update-package jquery –version 1.9.1
PM> update-package knockoutjs
```

Next we'll add a JS called knockout.protectedObservable.js and add the following script by Ryan Niemeyer.

```
// Source-Ryan Niemeyer's JS Fiddle at http://jsfiddle.
net/rniemeyer/X9rRa/
```

```
//wrapper for an observable that protects value until
committed
ko.protectedObservable = function (initialValue)
{
 //private variables
 var _temp = initialValue;
 var _actual = ko.observable(initialValue);
 var result = ko.dependentObservable({
  read: _actual,
  write: function (newValue)
  {
   _temp = newValue;
  }
 });
 //commit the temporary value to our observable, if it
is different
 result.commit = function ()
 {
  if (_temp !== _actual())
  {
   _actual(_temp);
  }
 };
 //notify subscribers to update their value with the
original
 result.reset = function ()
 {
  _actual.valueHasMutated();
  _temp = _actual();
 };
 return result;
};
```

The above script prevents Updates to the ViewModel unless we call an explicit commit() method. Once this script is in, our dependencies are set. Let's build a View Model.

THE VIEW MODEL

We'll add a new JavaScript file to the Scripts folder called **timekeepr-vm.js**

In the ViewModel, we have a collection called timecards that essentially lists all the time cards in the database.

The selectedTimeCard property is set when the user clicks on the 'Select' link on any of the Time Card rows OR when a new TimeCard is added.

The addNewTimeCard method is called when user clicks on the "Add New TimeCard" button. It adds an empty TimeCard object into the timecards collection. This method is also called when we are loading data from server into our ViewModel. In that case, it maps the JSON objects into KO Observable objects.

The commitAll function is a helper function that commits each property in the selectedTimeCard. It has to be called before we save any changes to the database.

```
var viewModel = {
 timecards: ko.observableArray([]),
 selectedTimeCard: ko.observable(),
 addTimeCard: function ()
 {
  viewModel.timecards.push(addNewTimeCard());
 },
 selectTimeCard: function ()
 {
  viewModel.selectedTimeCard(this);
 },
 commitAll: function ()
 {
  viewModel.selectedTimeCard().Summary.commit();
  viewModel.selectedTimeCard().Description.commit();
  viewModel.selectedTimeCard().StartDate.commit();
  viewModel.selectedTimeCard().EndDate.commit();
  viewModel.selectedTimeCard().Status.commit();
 }
};

function addNewTimeCard(timeCard)
{
 if (timeCard == null)
 {
  return {
   Id: ko.protectedObservable(0),
   Summary: ko.protectedObservable("[Summary]"),
   Description: ko.protectedObservable("[Description]"),
   StartDate: ko.protectedObservable((
                                new Date()).toJSON()),
   EndDate: ko.protectedObservable((
                                new Date()).toJSON()),
   Status: ko.protectedObservable("Tentative")
  };
 }
 else
 {
  return {
```

```
    Id: ko.protectedObservable(timeCard.Id),
    Summary: ko.protectedObservable(timeCard.Summary),
    Description: ko.protectedObservable(timeCard.
                                        Description),
    StartDate: ko.protectedObservable((new Date(timeCard.
                                StartDate)).toJSON()),
    EndDate: ko.protectedObservable((new Date(timeCard.
                                EndDate)).toJSON()),
    Status: ko.protectedObservable(timeCard.Status)
   };
  }
}
```

THE VIEW

Once the view model is in place, we setup our view by updating the Index.cshtml. This page is served up by the HomeController, but we don't have any Controller side code for it. We'll make AJAX calls to the API controller to manipulate data.

```
<div id="body">
 <section class="content-wrapper main-content clear-
fix">
  <div class="left-section">
   <div>
    <button id="addNewTimeCard" data-bind="click:
            addTimeCard">Add New TimeCard</button>
   </div>
   <table>
    <thead>
     <tr>
      <td>Summary</td>
      <td>Start Date</td>
      <td>End Date</td>
      <td></td>
     </tr>
    </thead>
    <tbody data-bind="foreach: timecards">
    <tr>
     <td data-bind="text: Summary"></td>
     <td data-bind="text: StartDate"></td>
     <td data-bind="text: EndDate"></td>
     <td><a href="#" data-bind="click: $parent.
               selectTimeCard">Select</a></td>
     <td><a href="#" class="downloadButton">Demo
                           VCalendar</a></td>
    </tr>
   </tbody>
  </table>
 </div>
 <div class="right-section" data-bind="with: selected-
TimeCard">
  <div>Summary</div>
 <div>
  <input data-bind="value: Summary" /></div>
 <div>Description</div>
  <div>
   <input data-bind="value: Description" /></div>
   <div>Start Date</div>
   <div>
    <input data-bind="value: StartDate" /></div>
   <div>End Date</div>
   <div>
    <input data-bind="value: EndDate" /></div>
   <div>Status</div>
   <div>
    <input data-bind="value: Status" /></div>
   <button id="submitButton">Save</button>
   <button id="cancelButton">Cancel</button>
  </div>
 </section>
</div>
@section Scripts{
 <script src="~/Scripts/knockout-2.2.1.debug.js">
 </script>
 <script
  src="~/Scripts/knockout.mapping-latest.debug.js">
 </script>
 <script
   src="~/Scripts/knockout.protectedobservable.js">
 </script>
 <script src="~/Scripts/timekeepr-vm.js"></script>
 <script src="~/Scripts/timekeepr-client.js"></script>
}
```

We have split the view into two using the 'left-section' and 'right-section' CSS classes. The 'left-section' has the List of all the TimeCards whereas the 'right-section' shows the currently selected card if any.

To edit a card, we 'Select' it first. This populates the selectedTimeCard property in the view model which is bound to the 'right-section' and thus it comes up automatically. Once we do the changes and save, the data is posted back to the server and the selectedTimeCard is cleared out.

The implementation for adding new, selecting and saving data is in the timekeeper-client.js.

```
$(document).ready(function ()
{
 $.ajax({
  url: "/api/TimeCard",
  method: "GET",
  contentType: "text/json",
  success: function (data)
  {
   viewModel.timecards.removeAll();
   $.each(data, function (index)
   {
    viewModel.timecards.push(addNewTimeCard(
        data[index]));
   });
   ko.applyBindings(viewModel);
  }
 });
```

Once the document loads, we call the server to get all the time cards available. When the server returns the data, we add each TimeCard to the view model. Since the 'timecards' property is a KO Observable, Knockout will automatically update the view as each timecard is added.

```
 $(document).delegate("#submitButton", "click", function
()
 {
  viewModel.commitAll();
  var vm = viewModel.selectedTimeCard;
  var current = ko.utils.unwrapObservable(vm);
  var stringyF = JSON.stringify(ko.mapping.
toJS(current));
  var action = "PUT";
  var vUrl = "/api/TimeCard?Id=" + current.Id();
  if (current.Id()==0)
  {
   action = "POST";
   vUrl = "/api/TimeCard";
  }
  $.ajax(
  {
   url: vUrl,
   contentType: "application/json;charset=utf-8",
   type: action,
   data: stringyF,
   success: function (response)
   {
    alert("Saved Successfully");
    viewModel.selectedTimeCard(null);
   },
   failure: function (response)
   {
    alert("Save Failed");
   }
  });
 });
```

The Submit button click handler has the implementation for sending the currently selected TimeCard's information to be saved in the Server.

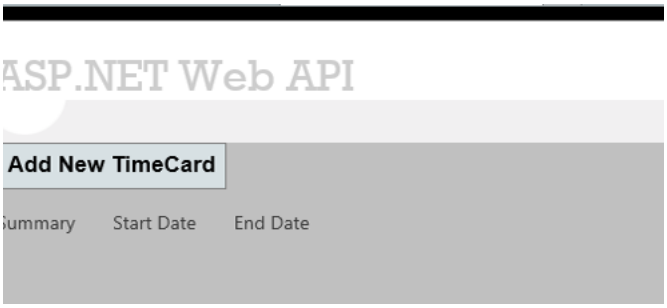
To do this, the 'selectedTimeCard' object is converted into a JSON string and put in the payload of the Ajax POST.

```
$(document).delegate("#cancelButton", "click", function
()
{
 viewModel.selectedTimeCard(null);
});
```

Cancelling the Edit process simply resets the selectedTimeCard property to null. This hides the Edit Panel on the UI.

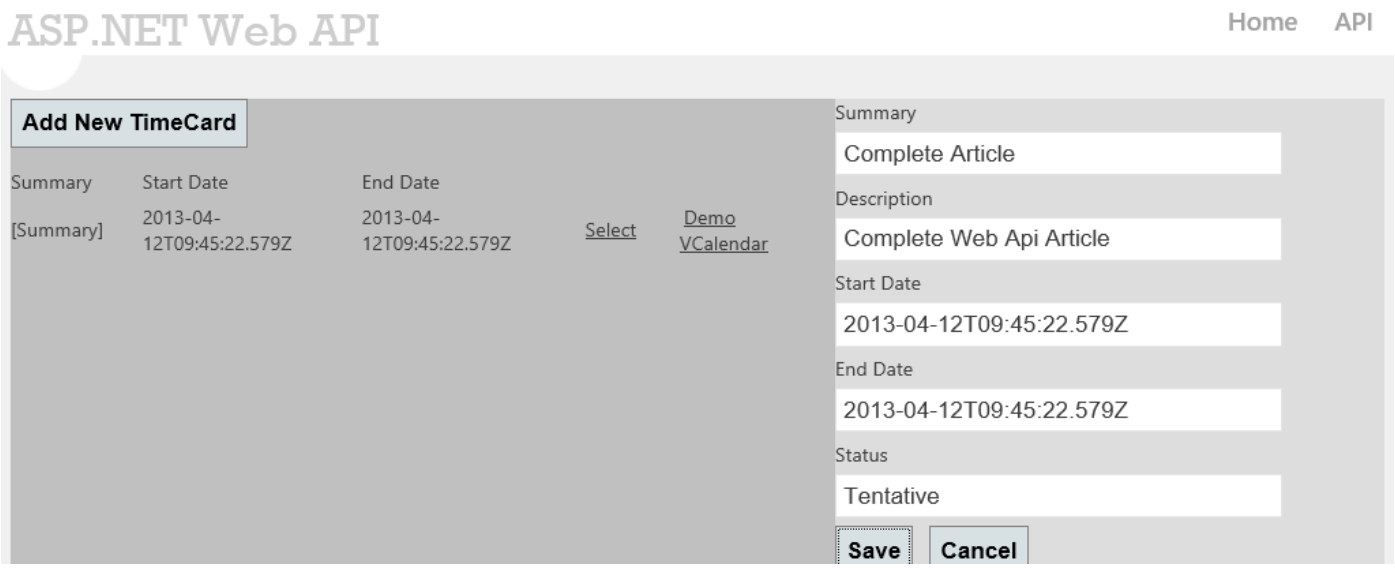
RUNNING THE APP TO GET AND PUT/POST DATA TO WEB API

Now that we have our Web API Controller and Index.cshtml with its client side dependencies setup, let's run the app. On first run, we are greeted with a near empty screen with only the 'Add New TimeCard' button and the list headers for the TimeCards

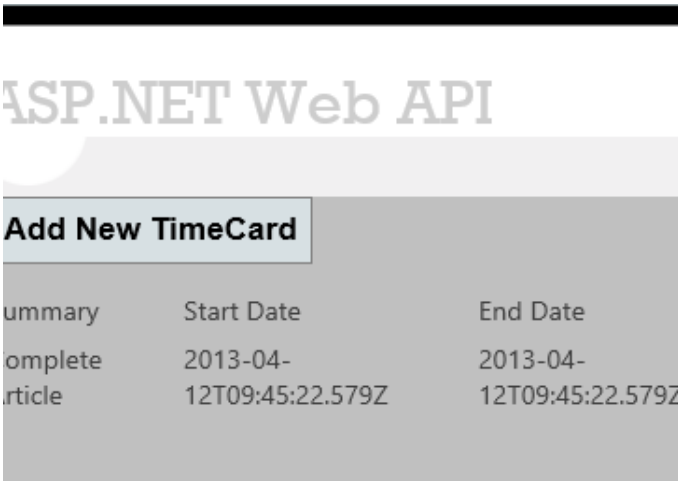


Clicking on 'Add New TimeCard' adds a default time card

We can edit this by clicking on the 'Select' link

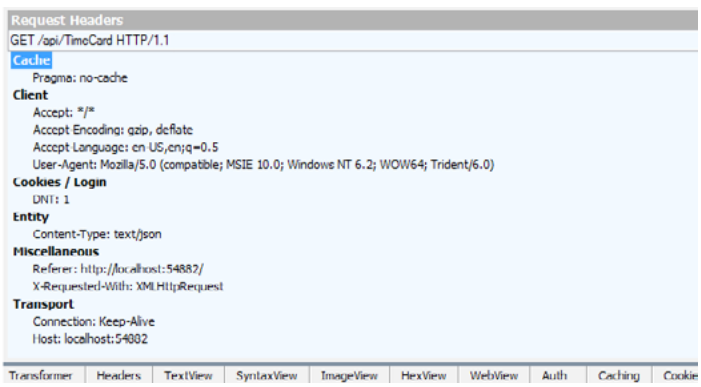


We update the Summary and Description before hitting 'Save', the data gets updated in the DB

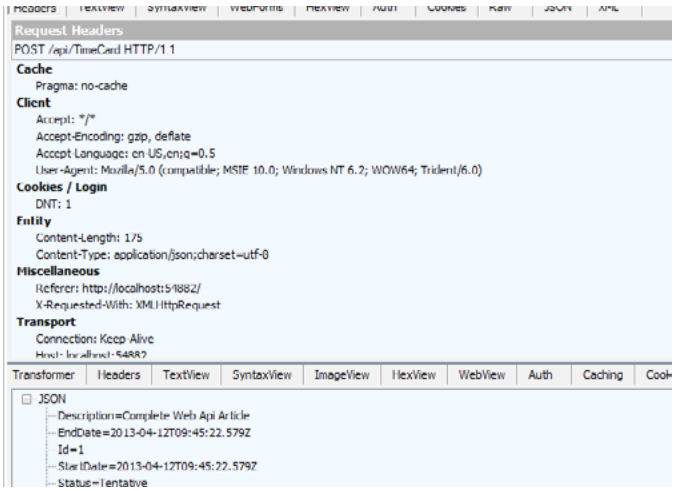


Behind the Scenes

Now that we've seen what's happening in the UI, let's see what's happening behind the scenes. To look at the HTTP Traffic, I had kept Fiddler running and the first request in Fiddler is a GET request which as we can see below, gets an empty JSON in return.



The next POST request pushes JSON Data to the Server.



On the next load, we again do a GET request and we see an array of JSON objects is returned, it has the newly created object in it.

In each of the cases, if you see the 'Content-Type' header parameter, it's set to Content-Type: text/json because we are requesting for the JSON data or posting JSON data. The Web API service can serve JSON and XML, it sees a request for JSON and uses the JSON formatter to convert the Entity or Entity List into JSON. It's worth noting that in our API Controller, we return Domain entities or their lists directly. Note the GetTimeCards and the GetTimeCard methods below.

```
namespace TimeKeepr.Models
{
 public class TimeCardController : ApiController
```

```
 {
  private TimeKeeprContext db = new TimeKeeprContext();
  // GET api/TimeCard
  public IEnumerable<TimeCard> GetTimeCards()
  {
   return db.TimeCards.AsEnumerable();
  }
  // GET api/TimeCard/5
  public TimeCard GetTimeCard(int id)
  {
   TimeCard timecard = db.TimeCards.Find(id);
   if (timecard == null)
   {
    throw new HttpResponseException(Request.
CreateResponse(HttpStatusCode.NotFound));
   }
   return timecard;
  }
  …
 }
}
```

CUSTOM MEDIA TYPE FORMATTERS AND CONTENT TYPES

Thus, we've seen how the out of the box media formatters work, let's see what options we have for formatting data with a custom media formatter.

USE CASE

The information we have can be easily saved as Calendar items. So it would be nice if our service could serve up VCalendar items too. VCalendar is a data format that most calendaring tools support. If our Web API could serve up VCalendar items for each TimeCard, then they could be easily integrated with calendaring tools like Outlook.

With this use case in mind, let's create a VCalendarMediaTypeFormatter.

OUR CUSTOM MEDIA TYPE FORMATTER

Custom Media Type Formatters subclass the MediaTypeFormatter abstract class or one of its implementations. We will subclass the BufferedMediaTypeFormatter.

```
public class VCalendarMediaTypeFormatter :
BufferedMediaTypeFormatter
{
 …
}
```

First thing we do in the constructor is to clear out all Supported media types and only keep "text/calendar", which is the standard header type for requesting calendar items.

```
public VCalendarMediaTypeFormatter()
{
 SupportedMediaTypes.Clear();
 SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/
calendar"));
}
```

Next we override the SetDefaultContentHeaders and set content-disposition as an attachment with a generic file name (myFile.ics).

```
public override void SetDefaultContentHeaders(Type type,
HttpContentHeaders headers, MediaTypeHeaderValue
mediaType)
{
 headers.Add("content-disposition", (new
    ContentDispositionHeaderValue("attachment") {
  FileName = "myfile.ics" }).ToString());
 base.SetDefaultContentHeaders(type, headers,
                                    mediaType);
}
```

Since our formatter will not be accepting ICS files in posts, we'll override the CanReadType method and return false directly.

```
public override bool CanReadType(Type type)
{
 return false;
}
```

Our Formatter should only try to format requests of type TimeCard and not any other type. So we limit this in the CanWriteType method by checking if the incoming type is of type TimeCard only.

```
public override bool CanWriteType(Type type)
{
 if (type == typeof(TimeCard))
 {
  return true;
 }
}
```

Next comes the crux of the formatter - the WriteToStream method. We cast the incoming object into a TimeCard and call the WriteVCalendar method that writes to the stream buffer.

```
public override void WriteToStream(Type type, object
value, Stream stream, System.Net.Http.HttpContent
content) {
 using (var writer = new StreamWriter(stream))
 {
  var timeCard = value as TimeCard;
  if (timeCard == null)
  {
   throw new InvalidOperationException("Cannot serialize
type");
  }
  WriteVCalendar(timeCard, writer);
 }
 stream.Close();
}
```

As we can see below, VCalendar items are simply formatted text and we use the data in our TimeCard to create the appropriate string for the VCalendar.

```
private void WriteVCalendar(TimeCard contactModel,
StreamWriter writer) {
 var buffer = new StringBuilder();
 buffer.AppendLine("BEGIN:VCALENDAR");
 buffer.AppendLine("VERSION:2.0");
 buffer.AppendLine("PRODID:-//DNC/DEMOCAL//NONSGML v1.0//
EN");
 buffer.AppendLine("BEGIN:VEVENT");
 buffer.AppendLine("UID:suprotimagarwal@example.com");
 buffer.AppendFormat("STATUS:{0}\r\n", contactModel.
Status);
 buffer.AppendFormat("DTSTART:{0}Z\r\n", (contactModel.
StartDate.ToFileTimeUtc().ToString()));
 buffer.AppendFormat("DTEND:{0}Z\r\n", (contactModel.
EndDate.ToFileTime().ToString()));
 buffer.AppendFormat("SUMMARY:{0}\r\n", contactModel.
Summary);
 buffer.AppendFormat("DESCRIPTION:{0}\r\n", contactModel.
Description);
 buffer.AppendLine("END:VEVENT");
 buffer.AppendLine("END:VCALENDAR");
 writer.Write(buffer);
}
```

ASSOCIATING CUSTOM FORMATTER IN OUR WEBAPI APPLICATION

This is a single line of configuration in code. In the App_Start\WebApiConfig.cs, we add the following in the last line of the Register method

```
config.Formatters.Add(new VCalendarMediaTypeFormatter());
```

If you don't have the WebApiConfig.cs, you can use

```
GlobalConfiguration.Configuration.Formatters.Add(new
VCalendarMediaTypeFormatter());
```

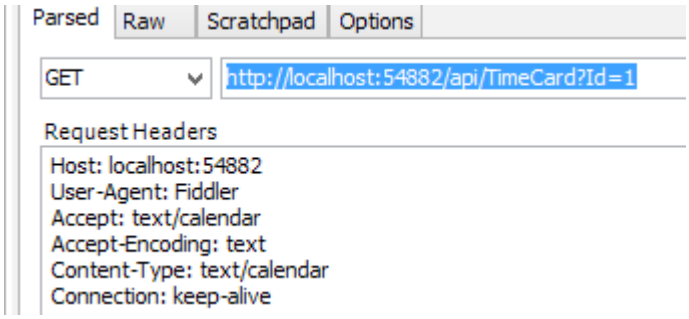
That's it we are done.

TESTING THE CUSTOM FORMATTER OUT

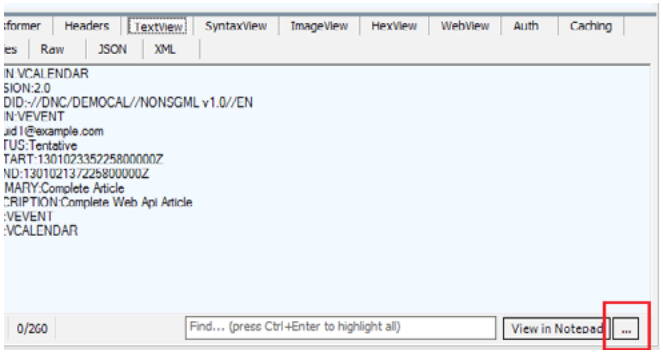
Our Formatter is all set, how can we test this? Well we can test it using Fiddler by creating a new Request as follows. In the Composer Tab, we set the Request Type to a GET put the following headers

Host: localhost:54882
User-Agent: Fiddler
Accept: text/calendar
Accept-Encoding: text
Content-Type: text/calendar
Connection: keep-alive

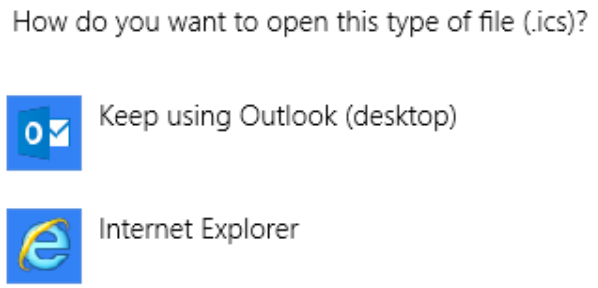
We post it to the URL
http://localhost:54882/api/TimeCard?Id=1



Once the post completes, we switch to the Inspectors and select 'TextView'. We can see the Text returned by our Web API service. It detected the Content-Type request and sent us back a 'text/calendar'. To get the actual file, click on the ellipsis button to the right of 'View in Notepad' button, at the bottom of the screen.



On clicking the ellipsis, we get the following (on Windows 8). You will see all apps that are registered to deal with ICS files in that list



On selecting Outlook, I get the Calendar item imported into my calendar. Check the screenshot below!

Isn't that awesomesauce?

We just built a Custom Media Type Formatter that can format our TimeCard data into a VCalendar that can be downloaded and imported into Outlook!

TO WRAP UP

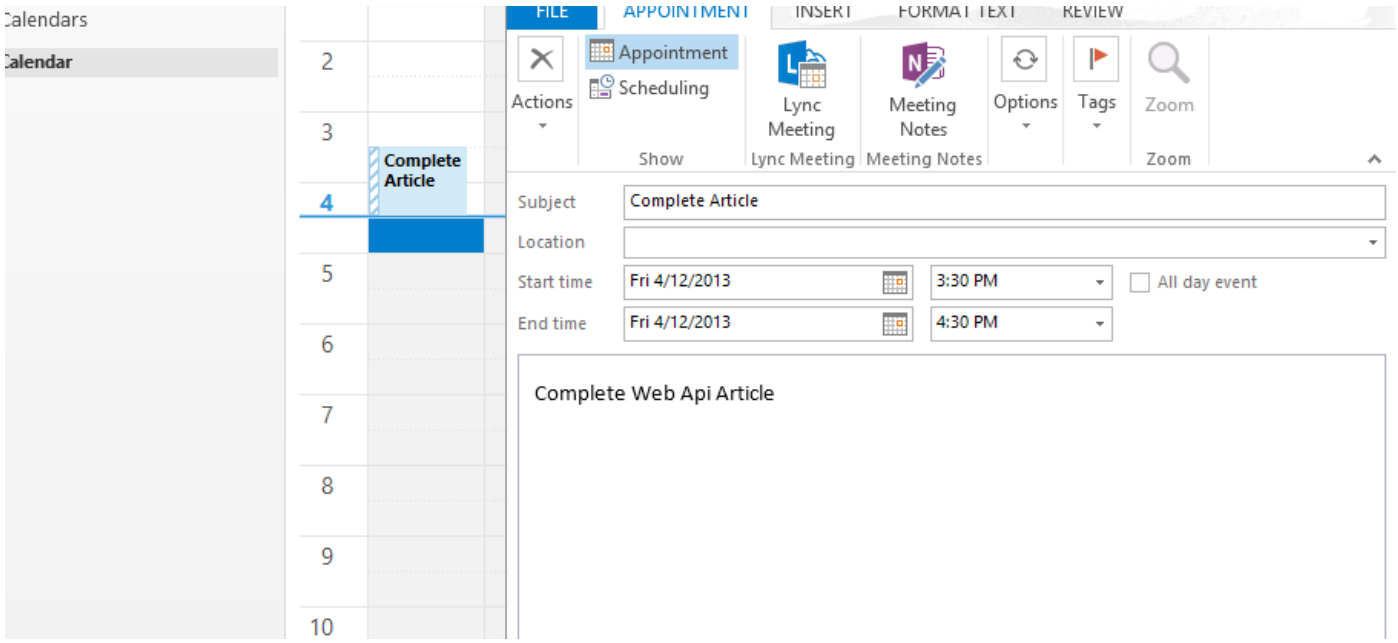
Well, with the Media Type Formatter demo, we bring this article to a close. Today we saw how to build web services using ASP.NET WebAPI and access them over HTTP using AJAX. We sent our service request via browser as well as via a HTTP tool called Fiddler. We got back response as JSON and we built a custom formatter to get data as a VCalendar.

Now that we have our service in place, we can very easily build clients on any platform that can communicate over HTTP (which is essentially all platforms). So we could have Windows 8 App or a Windows Phone App or an iOS app talk to the same service and get back this same data.

We covered two key areas of Web API. We have not looked at Authentication and Authorization among other things. Those we leave for another day ■

Download the entire source code from Github at bit.ly/dncm6-wapicf

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



Push Notifications in **Windows 8 Apps**

Sumit Maitra introduces the Windows Push Notification Service and how to implement it for Windows 8 Store Apps using Azure Mobile Services

With the popularity of Smart Phones and always connected (to internet) devices, 'push notifications' have gained immense popularity. Though push notifications are not new and Exchange had implemented Active Sync quite some time back, smartphones have made them ubiquitous and brought them out of the Enterprise-y setups. So today, we often receive updates instead of looking (polling) for updates. This can be a more efficient process depending on the implementation.

Common examples of Push notification usage are, mail clients that receive new mail notifications as they come to their server's Inbox. E.g. Social Media apps that push updates to all subscribed as soon as someone makes an update to their status or games that can push leaderboard updates of your position changes and so on.

Given the popularity and utility of Push Notifications, it's a desired feature to have in mobile applications. However, setting up the infrastructure to do so is non-trivial on any client (Windows Phone, iOS, Android or Windows 8) platform. This is where the Windows Azure Team has stepped in with the Windows Azure Mobile Services that among other things, aim to ease the burden of building Push Notification infrastructure for connected apps.

Today, we will see how to build a Windows 8 Store App that is capable of receiving push notifications. For the backend infrastructure we will use Windows Azure Mobile Services.

Windows (Push) Notification Service

The underlying infrastructure that enables Push notifications is referred to as Windows Push Notification Service or WNS for short. The typical process is as follows:

1. You first get an App SID and Client Secret for your Windows 8 Store App and store it in your Web Service.

2. Next your App requests Windows for a 'Channel' and sends it to your WebService so that it can track the channels.

3. When one of the clients submit some data to your web service that needs to send out push notifications, the web service provides WNS with the SID and Client secret along with the channel details for the notification to propagate.

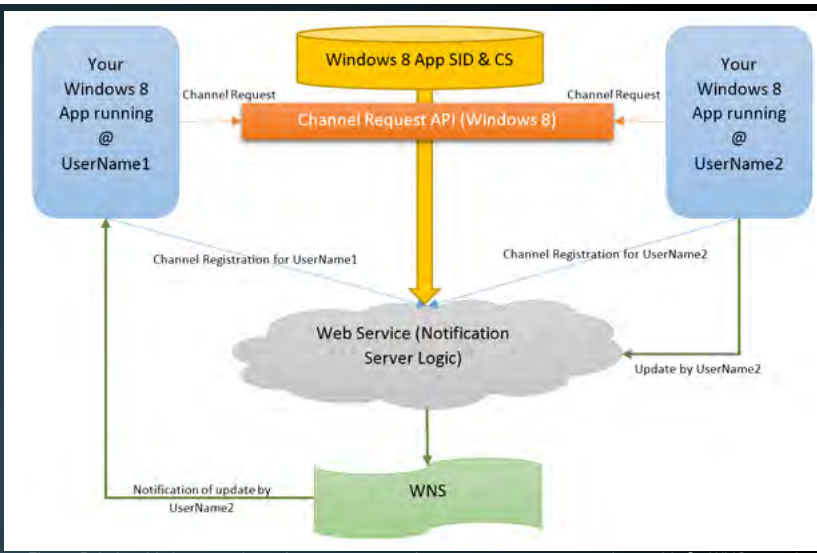


Figure: Relationship between the various components that your app must communicate with, for WNS to work

4. All clients that the data was intended for, then receive the push notification. The clients could be direct recipients or subscribed for by the client

5. Notifications in Windows 8 can be
   a. Tile notifications (primary/secondary)
   b. Toast notifications (for toast popup)
   c. Raw notifications (to download data)

AZURE MOBILE SERVICES AND PUSH NOTIFICATIONS

Azure mobile services step in and help developers by simplifying the Web Service/Notification Server logic. Azure Mobile Services provide the following for us:

1. An Entity Store for easy data access and storage. It uses Azure SQL DB as its backing store.
2. A client side communication library so you don't have to deal with Raw HTTP Requests.
3. Secure Storage for App SID and Client Secret (these should not be distributed with the clients).
4. Communication with WNS.

This significantly eases our task of building and managing Push notification service for our Mobile Apps.

THE USE CASE

Ideally we would like to build a Twitter client with push notifications, but that is a non-trivial example that can nearly span a book much less an article. So today, we'll build a fake Twitter application. Something, that is distantly similar to Twitter, only if you look at it with a very obtuse angle. We'll call this app ZumoFakeTweet!

Application Design

It will be a simple app that accepts a User Name identifying the user of the App. The premise of the App is to send out FTs (fake tweets) and FDMs (fake Direct Messages).

FTs are pushed to everyone whereas FDMs are pushed to the user whom we intend to send it.

But First a set of Pre-Requisites

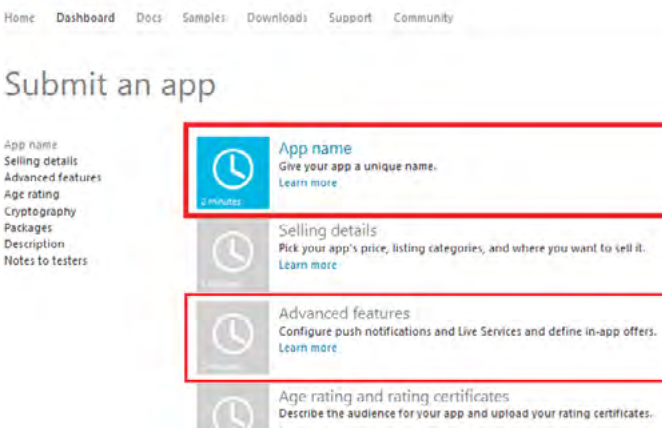
1. You'll need an active Azure Account and even though Mobile Services are in Preview and the first 10 are free, you'll still need an Azure SQL Database that may cost you depending on your plan.
2. You will need a Windows 8 Developer account to follow through with the Push Notification Setup and implementation.
3. Access to a Windows 8 machine with Visual Studio (Express will do).

With the basic premise set, let's look at the pre-requisites and go ahead and build the Windows 8 Application.

BUILDING 'ZUMOFAKETWEET'

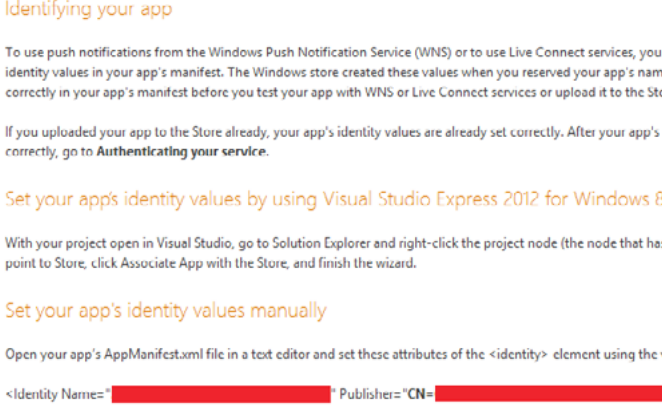
Configuring Windows 8 App, Azure Mobile Service and WNS

1. We start off by Logging into our Developer Dashboard and creating a new Application. To do so, you have to click on the 'Submit an app' menu item on the left hand menu bar.



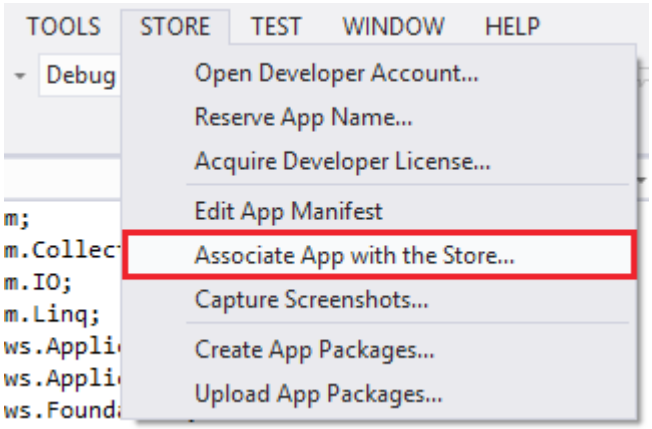
2. Click on the App name and provide the name of your App. If you already have an App that you want to add notification capabilities to, select 'Advanced Features'. We provide the name 'ZumoFakeTweet' and move on to the 'Advanced Features'.

3. In 'Advanced Features' page, click on the 'Push Notifications and Live Connect Info Service'. This pops up a new browser (you may have to re-login with your Windows Store developer account credentials). The First Step here is to 'Identify the Application'. As you can see below, you can either do it via Visual Studio or do some manifest file hacking later. We will create our Windows 8 Store App at this point.

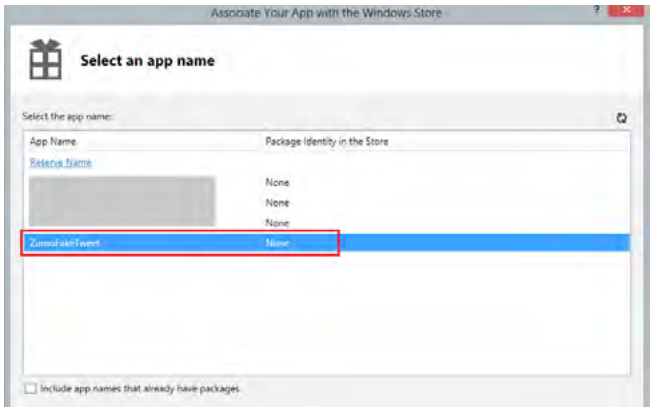


4. To get started with the ZumoFakeTweet project, start Visual Studio, select new Windows 8 Project and pick a Blank Template

5. Once Visual Studio is done setting up the boilerplate, build the project. After build succeeds, go to the 'Store' > Associate App With Store' menu item



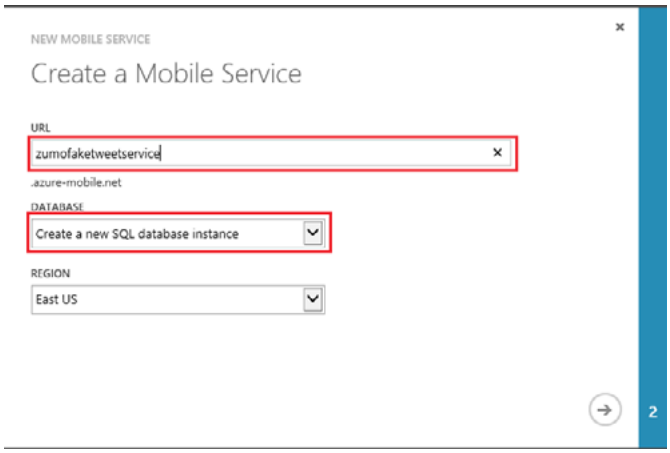
6. You will have to authenticate with your 'Windows 8 Store Developer Dashboard' credentials. Once logged in, you'll see a list of Apps that you have registered. It will include the 'ZumoFakeTweet' app. Select it and click Next. It will show you the association details. Click Finish to complete the process



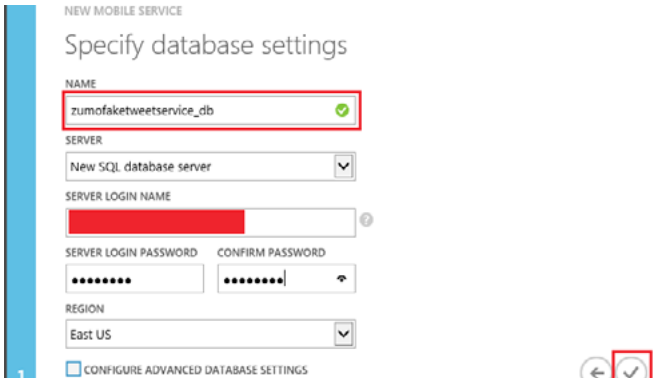
7. Now we'll jump back to the Authentication process in the browser and click on the 'Authenticating Your Service' link. This will give us the Package Security Identifier (SID) and Client Secret as shown below. As you can see, these two strings are supposed to be secret and not shared with anyone. Make a note of these two strings before we move on.

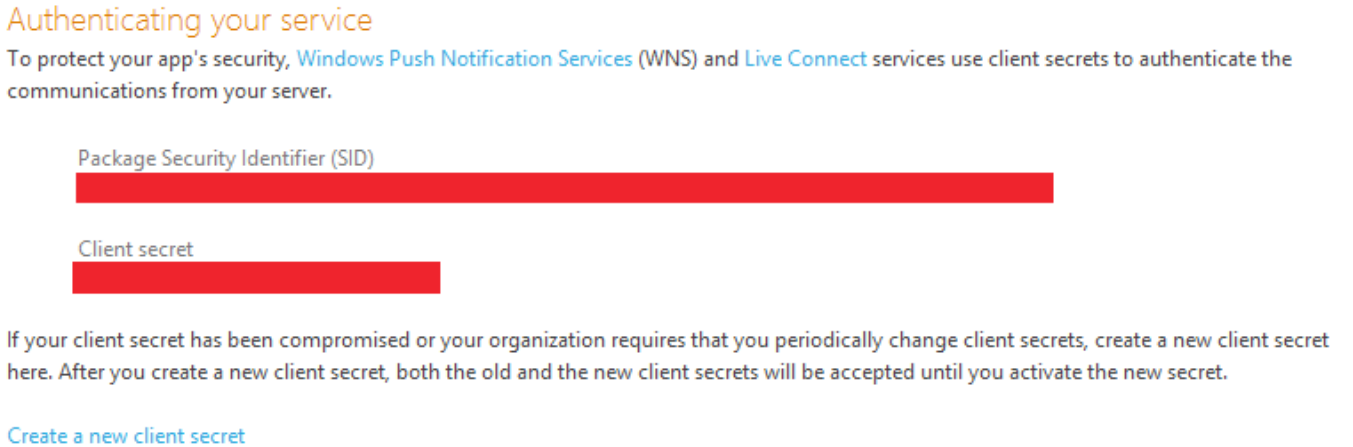
8. Time to get Azure into the mix. Log on to the Azure Management Portal with your Microsoft Account.

9. Click on Mobile Services and create a New Mobile Service. You have to pick a ServiceUrl. Whatever name you specify, the service will be available at http://[yourname].azure-mobile.net/
If you have SQL Databases in Azure already, you'll get to pick one or create a new one. I didn't have any, so in the next steps, I'll create a new one.

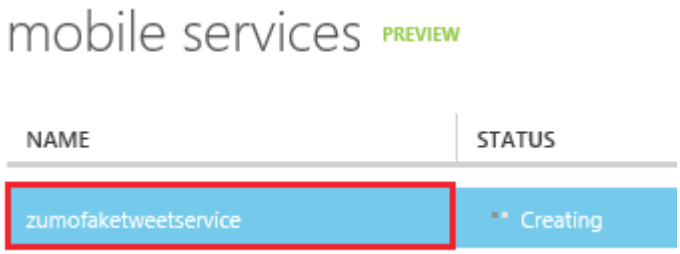


10. Click next to fill in the DB Details

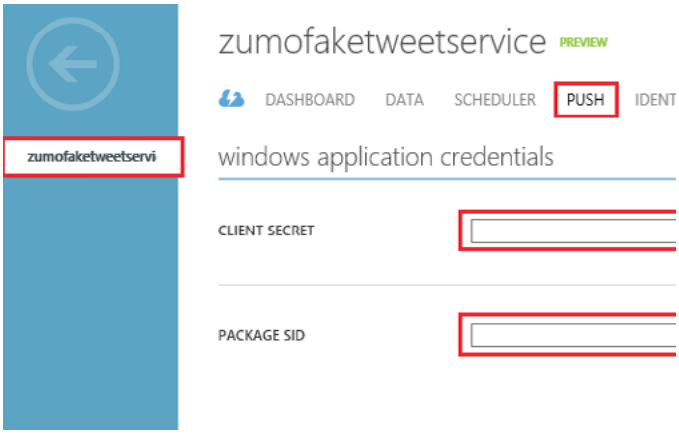




Provide a Database Name, an Admin User and Password. Select a Region that is closest to where you anticipate most traffic will come from. Click on the 'Complete' button to finish the Wizard. Azure will initiate the Service creation process.



11. Once the service is ready, click on it to navigate to the Dashboard. In the dashboard, select the 'PUSH' tab to configure push notifications. For Windows Application Credentials, provide the Client Secret and Package SID we noted down earlier. Click on 'Save' at the bottom of the screen to save the changes immediately.



Your App as well as Mobile Service is now set to work with WNS.

Building the Messaging Functionality

Now that we are all set, quick recap of what needs to be done from here on

1. Setup our App to have three fields,
   a. MyId – Current User's Id
   b. TweetText – Text of the FT or the FDM
   c. ToID – An optional ID to send FDMs to.

2. Save the above data in Azure Database. We'll use Azure Mobile Services for the same.

3. Write Code to retrieve Channel IDs

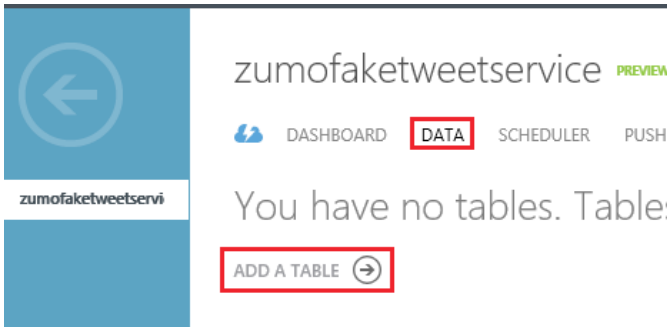
4. Utilize Channel IDs to send Push Notifications to others either selectively or via a broadcast.

Setting up the UI

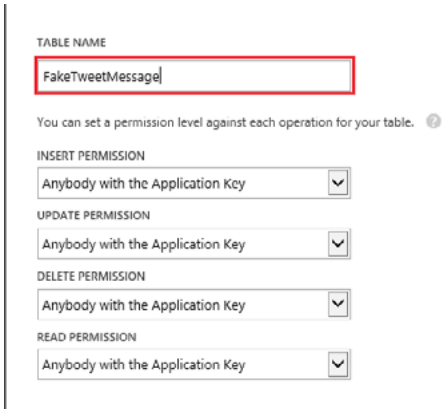
The UI is very simple and won't be winning any App Design contests. All it has is three labels, three text boxes and a Send Message Button.

Sending and Saving Data to Azure Mobile Service

Now that we have the UI setup, let's work on saving data. To do this, we have to switch back to the Mobile Service Dashboard.



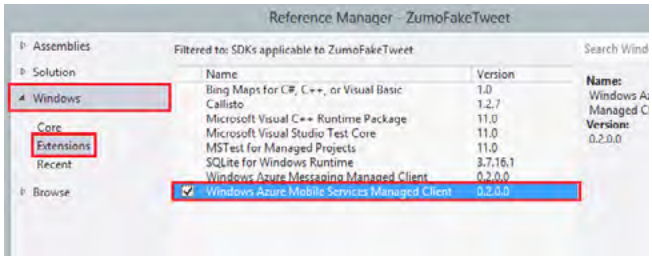
1. Add the FakeTweetsMessage Table and click OK. The columns will get created automatically once we post our Entity to it.



2. Before we can save the data, we need to add reference to 'Windows Azure Mobile Services Managed Client'. This is a simple wrapper to make HTTP calls easy for you. We already had it specified as a pre-requisite, so if you didn't have it installed yet, go grab either from Nuget now and add reference to it in the project. You can get it from Nuget as follows

```
PM> Install-Package WindowsAzure.MobileServices -Version 0.2.0
```

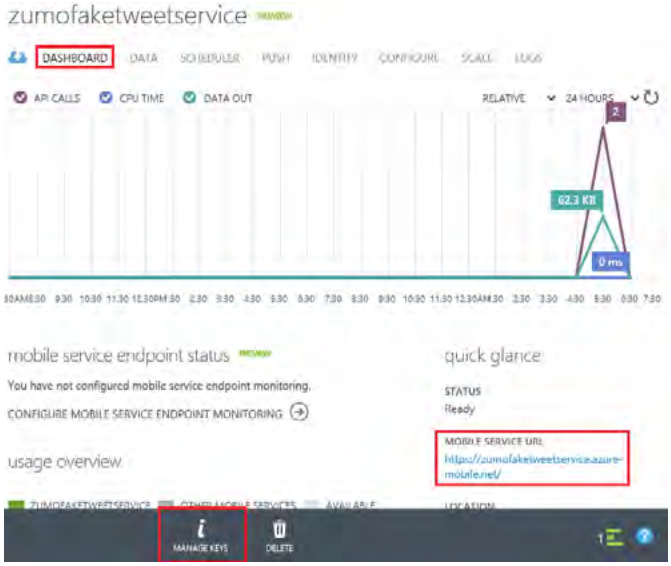
Or if you install the VSIX, you can add a reference as follows



3. To save data into the above table, we create a FakeTweetMessage class

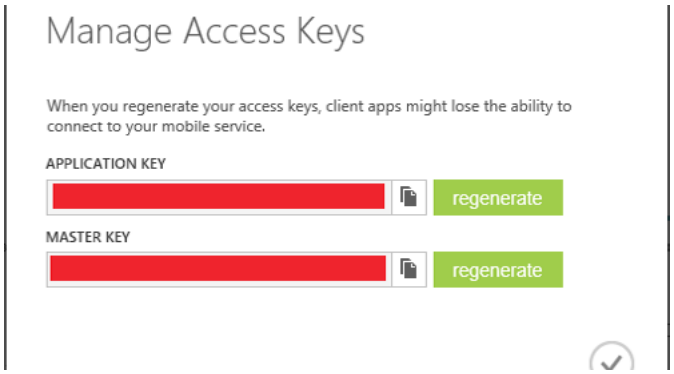
```
class FakeTweetMessage
{
 public int Id { get; set; }
 [DataMember(Name = "myId")]
 public string MyId { get; set; }
 [DataMember(Name = "fakeTweetText")]
 public int FakeTweetText { get; set; }
 [DataMember(Name = "toId")]
 public string ToId { get; set; }
}
```

4. Next we setup the Azure Client to talk to our Mobile Service. To do this, we first get the URL and the Application key from the Service Dashboard.



Note the Mobile Service URL before clicking on the 'Manage Keys' button

5. Clicking on Manage Keys will give us a popup that shows us the Application Key and Master Key. We need only the Application Key for now.



6. Posting Data to Azure Mobile Service

a. In the App.xaml.cs insert the following code

```
public static MobileServiceClient MobileService = new
MobileServiceClient(
  "https://zumofaketweetservice.azure-mobile.net/",
  "Replace with the Application Key from Above"
);
```

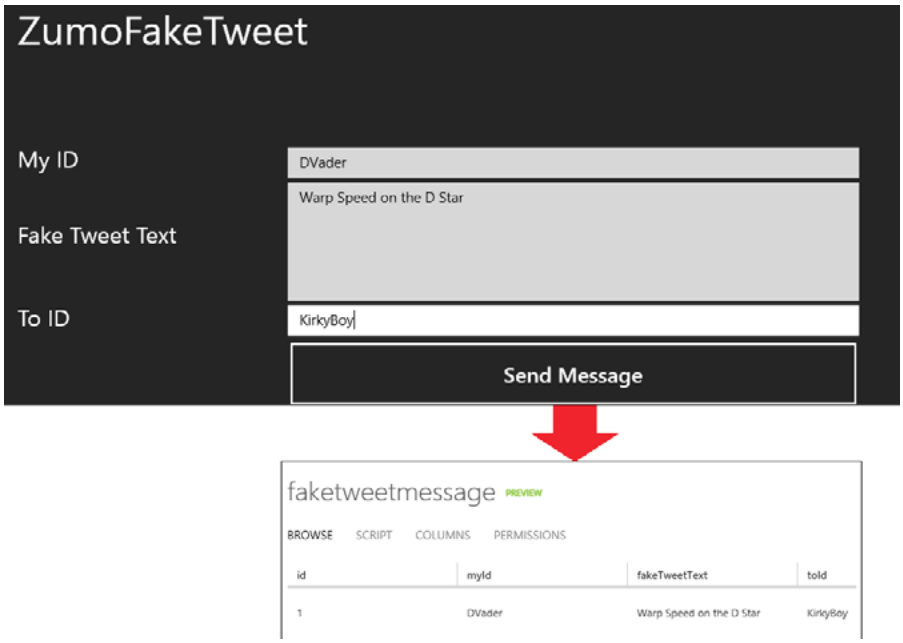
b. In the MainPage.xaml.cs, we will introduce the following:

```
private IMobileServiceTable<FakeTweetMessage>
fakeTweetMessages = App.MobileService.
GetTable<FakeTweetMessage>();
```

c. Add a Button Click handler for the Send Message button as follows

```
private async void SendMessageButton_Click(object sender,
RoutedEventArgs e)
{
 await fakeTweetMessages.InsertAsync(new FakeTweetMessage
 {
  MyId = MyIdText.Text,
  FakeTweetText = FakeTweetTextBox.Text,
  ToId = ToIdTextBox.Text,
 });
}
```

d. At this point, if we run the Application and provide some data in the Text Boxes and hit Send Message button, we'll end up with data in our Azure Table as follows.



So we have Data persistence in Azure Mobile Service working. But this is not triggering in any push notifications. We've some more work to do for that.

Sending Push Notifications to one User

1. Setting up a Channel Request - In App.xaml.cs, we add the following using statement

```
using Windows.UI.Xaml.Navigation;
```

Next we add the following code to it

```
public static PushNotificationChannel CurrentChannel {
get; private set; }
//WARNING: This should ideally be done once every
// 30 days when the channel expires.
private async void AcquirePushChannel()
{
 CurrentChannel =
await PushNotificationChannelManager.
CreatePushNotificationChannelForApplicationAsync();
}
```

Finally we invoke it in the OnLaunched event

```
protected override void OnLaunched(LaunchActivatedEventAr
gs args)
{
 AcquirePushChannel();
 …
}
```

So our app has now made a Channel request to WNS and has a new channel ready. Next step, is to let the server know.

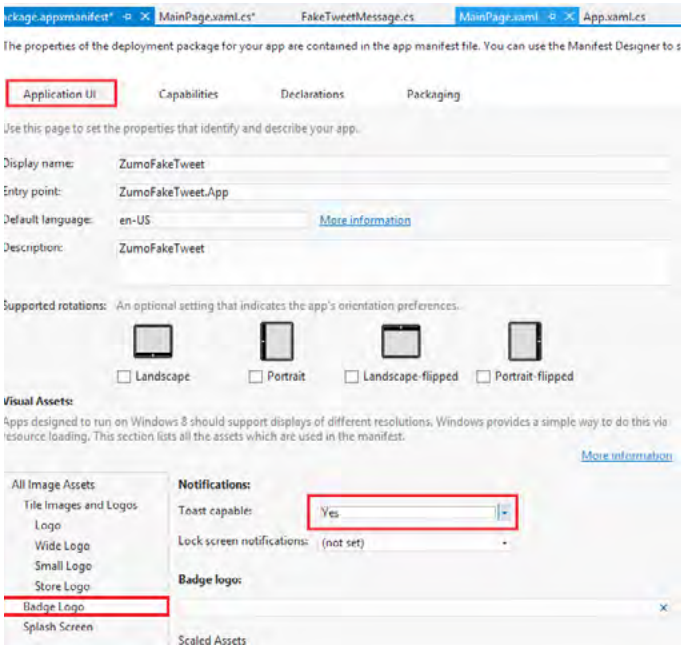
2. To pass channel information to the server, we need to expand our Entity by adding a Channel property to it.

```
[DataMember(Name = "channel")]
public string Channel { get; set; }
```

3. We populate this field on the Button Click event

```
private async void SendMessageButton_
Click(object sender, RoutedEventArgs e)
{
 await fakeTweetMessages.InsertAsync(new
FakeTweetMessage
 {
  MyId = MyIdTextBox.Text,
  FakeTweetText = FakeTweetTextBox.Text,
  ToId = ToIdTextBox.Text,
  Channel = App.CurrentChannel.Uri
 });
}
```

4. Finally let's enable Toast Notifications for our app. Open the Package.appxmanifest file and update the Badge Logo setting to set Toast Capable = Yes. This completes the client setup. Now we setup our Mobile service to intercept DB inputs and fire off notifications.



Intercepting DB Inputs and sending Notifications

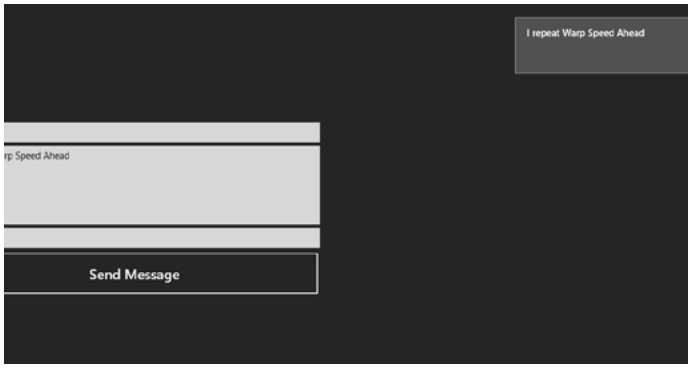
1. We open our 'FakeTweetMessage' database and select 'Script'. For the INSERT Script, we update the default code with the following

```
function insert(item, user, request) {
    request.execute({
        success: function() {
            request.respond();
            push.wns.sendToastText04(item.channel, {
                text1: item.fakeTweetText
            }, {
                success: function(pushResponse) {
                    console.log("Sent push:",
                                    pushResponse);
                }
            });
        }
    });
}
```

This is (if you haven't guessed already) JavaScript! On the Server, we define our post-Insert DB Logic in JavaScript.

On Success, the script uses the push.wns object to send a type of Toast (In this case Text 04). It uses the item.channel to set the target user to whom the notification is going to be sent. And text1 is the message to be sent.

Save this function and run the application. On hitting Send Message, you should receive a Toast Notification with the message you sent.



Congratulations, you just sent out the first Push Notification!

Sending Notifications Selectively

Now that we have seen how to lob back notifications to where it came from, let's look at a more practical scenario. This would be where we have multiple users signed up to receive push notifications. When a Message goes out with their Name in the To ID field, they should receive the message. Similarly if they subscribe for broadcasted messages, they should receive any message that does not have a To ID. Let's see what it takes to implement this simple scenario.

Adding Meta information to Channel Subscription

So far we've stuck Channel Subscription in our faketweetmessage table. Now we'll separate it out into a dedicated channel table with two more values

a. MyID – ID of the channel subscriber
b. AcceptBroadcast – A Boolean indicating if we want to subscribe to broadcasted messages.

The new Channel entity is as follows:

```
public class ChannelSub {
 public int Id { get; set; }

 [DataMember(Name = "channel")]
 public string Channel { get; set; }

 [DataMember(Name = "myId")]
 public string MyId { get; set; }

 [DataMember(Name = "acceptBroadcast")]
 public bool AcceptBroadcast { get; set; }
}
```

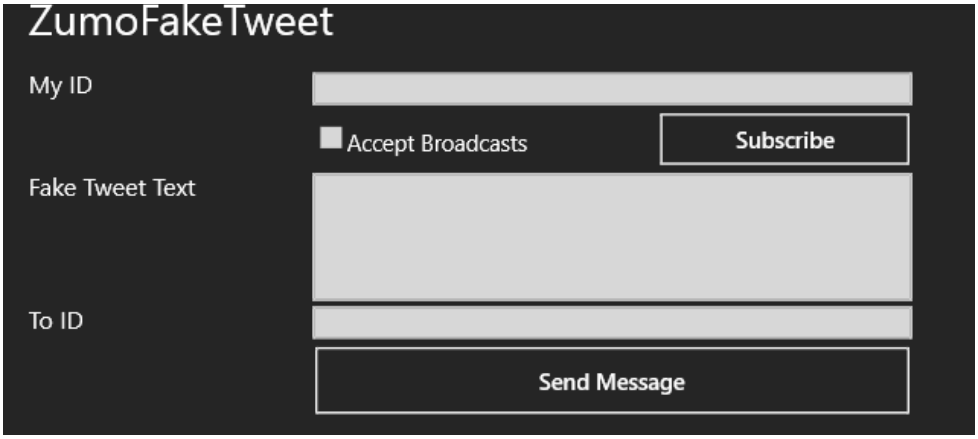
The Updated FakeTweetMessage is setup as follows

```
class FakeTweetMessage{
 public int Id { get; set; }

 [DataMember(Name = "myId")]
 public string MyId { get; set; }

 [DataMember(Name = "fakeTweetText")]
 public string FakeTweetText { get; set; }

 [DataMember(Name = "toId")]
 public string ToId { get; set; }
}
```

Updating the UI for Explicit Subscription

Now we'll update the UI by adding a 'Subscribe' button and a

Check Box for the AcceptBroadcast field. Our new UI will be as follows



Subscribing Channels

We'll put in a Clicked handler for the Subscribe button and update the client side code as follows:

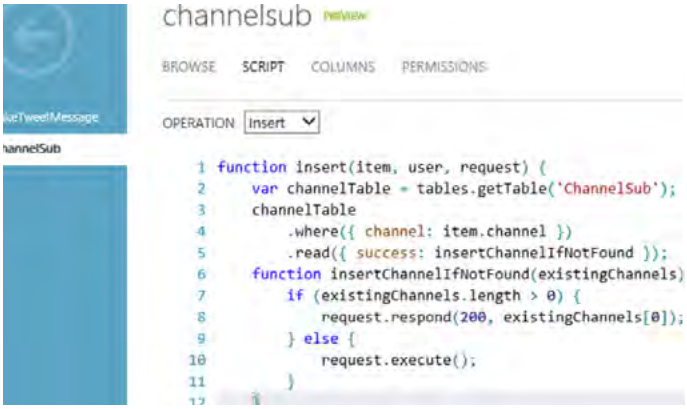
- In MainPage.xaml, we add the following statement to be able to access the new ChannelSub table

```
private IMobileServiceTable<ChannelSub> channelSubs =
App.MobileService.GetTable<ChannelSub>();
```

- The Subscription button click handler is as follows

```
private async void SubscribeButton_Click(object sender,
RoutedEventArgs e) {
 await channelSubs.InsertAsync(new ChannelSub
 {
  MyId = MyIdTextBox.Text,
  AcceptBroadcast = AcceptBroadcastCheckBox.IsChecked.
Value,
  Channel = App.CurrentChannel.Uri
 });
}
```

On the server, we'll have to setup the Insert script such that it doesn't allow duplicate subscription requests.



As we can see, we have added a new table called ChannelSub and added the following Insert Script.

```
function insert(item, user,
request) {
 var channelTable = tables.
         getTable('ChannelSub');
 channelTable
  .where({ channel:
                item.channel })
  .read({ success:
      insertChannelIfNotFound });
 function insertChannelIfNotFound
(existingChannels) {
  if (existingChannels.length >
0) {
   request.respond(200, existingChannels[0]);
  } else {
   request.execute();
  }
 }
}
```

The script checks the table to see if the requested Item's Channel URI already exists, if not, it inserts the new Channel Subscription requests.

Sending out Push Notifications based on Type of Subscription

Now that we have subscription information with us, let's modify the notification logic so that:

a. Users not subscribed to Broadcasts do not receive notifications without a To ID and the ones who are subscribed, get all requests

b. Instead of lobbing the Notification back to where it came from, if it has a To ID, send to that subscription ONLY.

To achieve the above two, we update the code with the Insert script of the FakeTweetMessage as follows:

```
function insert(item, user, request) {
 request.execute({
  success: function() {
   request.respond();
   sendNotifications();
  }
 });

function sendNotifications() {
```

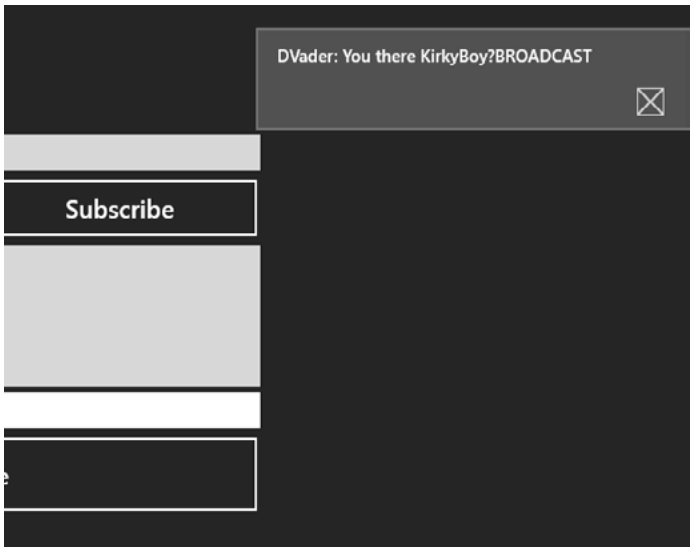
```
var channelTable = tables.getTable('ChannelSub');
channelTable.read({
success: function(channels) {
  channels.forEach(function(channel) {
   if((channel.acceptBroadcast && channel.myId != item.
      myId) ||
    (channel.myId==item.toId)){
    push.wns.sendToastText04(channel.channel, {
     text1: item.myId + ': ' + item.fakeTweetText +
     (item.toId === '' ? ' BROADCAST': '')
    }, {
        success: function(pushResponse) {
         console.log("Sent push:", pushResponse);
        }
    });
  } });
} }); }}
```

Here in the sendNotifications method, we are looping through all the subscribers and sending them a message if the Incoming request's 'toId' matches (and it's not the same as the sender) or if the channel subscribers are subscribed to receive broadcast messages.

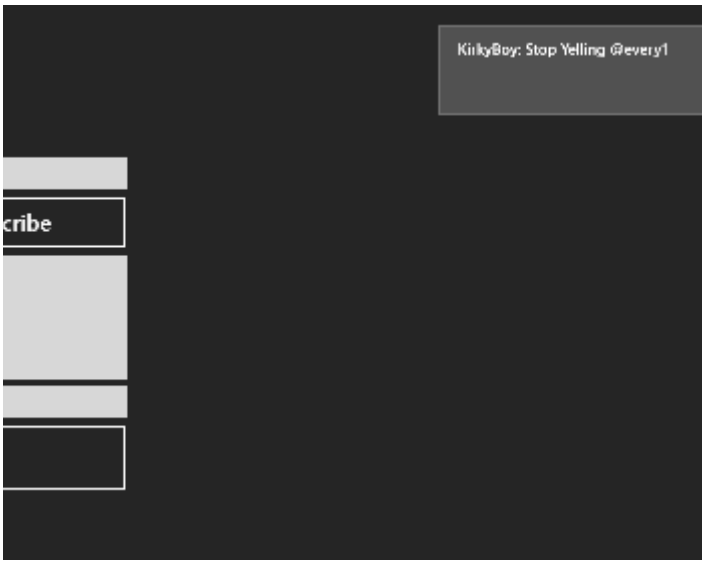
Demo

With the scripts in place, we load this application up on two machine, a Desktop and a Surface RT. On Desktop we have a user called DVader and on the Surface we have a user called KirkyBoy. Both are subscribed to receive broadcast messages.

Notification 1: DVader sends out a Broadcast message asking if KirkyBoy is around. KirkyBoy receives a toast notification for this as follows



Notification 2: KirkyBoy sees the 'BROADCAST' tag and is irked at 'DVader' for broadcasting a message and retorts back to DVader directly.



Nice eh!

WRAPPING IT UP

With that little squabble between our fictitious characters, we have seen Azure Mobile Service Notifications in full swing. It is quite amazing how the complex task of co-ordination between AppServer and WNS as well as registration and security ID were all handled by Azure Mobile services.

Going forth, be it social media apps or business apps, implementing Push notifications are really going to be a breeze and with the cost of one shared SQL Server DB instance, the amount of plumbing work saved, definitely offsets the expenditure of developing and maintaining a stable notification system from ground up ■

The complete code sample can be downloaded from bit.ly/dncm6-zutw but you do need to put in our Azure Mobile Service Application ID.

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