ISSUE 20 | SEP - OCT 2015

DNCMAGAZINE

www.dotnetcurry.com

Using
EDGE.JS
to combine
Node.js and
.NET

New
Build Features in
TFS 2015 and
Visual Studio Online

DEVOPS ON AZURE

DIAGNOSTIC
ANALYZERS
in Visual Studio 2015

OPEN CLOSED PRINCIPLE
Software Gardening:
Seeds

WPF ITEMS CONTROL
in depth

Digging Deeper Into
High-Trust Apps in
SharePoint

Building a Sound Cloud
Music Player in
Windows 10

Using
Bootstrap, jQuery and
Hello.js to integrate
Social Media
in your Websites

Using new XAML tools
in Visual Studio 2015

TABLE OF
CONTENTS

TEAM AND CREDITS

Editor In Chief
Suprotim Agarwal
suprotimagarwal@a2zknowledgevisuals.com

Art Director
Minal Agarwal
minalagarwal@a2zknowledgevisuals.com

Contributing Authors
Craig Berntson
Damir Arh
Edin Kapic
Gouri Sohoni
Irwin Dominin
Kent Boogaart
Mahesh Sabnis
Ravi Kiran
Shoban Kumar
Subodh Sohoni

Technical Reviewers
Kent Boogaart
Ravi Kiran
Suprotim Agarwal

Next Edition
2nd Nov 2015
www.dotnetcurry.com/magazine

POWERED BY

a2Z | Knowledge Visuals

FROM THE EDITOR

Suprotim Agarwal

Editor in Chief

We had a gala time in July releasing our 3rd Anniversary Edition. Overall it was a huge success with over 77K downloads. We also received tons of appreciation and feedback. In the next edition (November), you will see articles on React.js, Node.js and Angular.js as a direct result of this feedback.

With the Visual Studio 2015 and Windows 10 releases fresh in our mind; this month, we've put together four special articles on these technologies by Gouri, Mahesh, Shoban and first time DotNetCurry author Damir Arh. Welcome Damir!

Kent and Edin follow up on their previous articles on WPF ItemsControl and SharePoint High-Trust apps respectively, and go deep-dive in this edition. Subodh demonstrates how Azure supports the tasks of operations team as part of DevOps and Craig in his regular Software Gardening column covers the "O" (Open-Closed Principle) in SOLID.

Ravi introduces an open source project called Edge.js and showcases how Edge helps in integrating .NET and Node.js in your apps, while Irvin cooks up a nice little example of integrating Social Media with authentication in your websites.

THANK YOU for your feedback and please do continue sending them. We read each one of them and even incorporate the ones which are feasible at the moment. Reach out to us on twitter via our handle @dotnetcurry or mail me at **suprotimagarwal@dotnetcurry.com**.

ASP.NET MVC CONTROLS

WORK EFFORTLESSLY WITH ASP.NET MVC

Quickly create advanced, stylish, and high performing UIs for ASP.NET MVC with Ignite UI MVC. Leverage the full power of Infragistics' JavaScript-based jQuery UI/HTML5 control suite with easy-to-use ASP.NET MVC helpers and get a jump start on even the most demanding Web applications.

Download ASP.NET MVC Controls as part of the Ultimate Developer toolkit.

DOWNLOAD FREE TRIAL

INFRAGISTICS®

USING EDGE.JS to combine Node.js and .NET

> Today, we have an increasing number of technology platforms available to perform similar tasks. Each of these platforms are great in some areas though most of them are pretty close to each other when we take industry needs into consideration. It is always a challenge to make two different platforms work together. When we try to make it happen, we end up putting a lot of our time and energy in making the binaries of the platforms compatible, finding best ways to hook them together and deploying the application, using a hybrid platform setup. But if we have frameworks or, tools which does the heavy lifting of making the platforms work together, our lives would be much simpler.

Node.js is gaining a lot of popularity these days because it is unifying the language used on both server and client side. Though the platform proved that it can do almost everything that any other server platform based on established platforms like .NET and Java can do, it still lacks certain features that the other platforms do with ease. One of them is interacting with SQL databases. Though there are a few NPM packages to interact with SQL databases using Node.js, it is not easy to use them as they seem incomplete for the Node environment. Edge.js provides a solution to this problem by making it possible to call .NET functions from Node.js.

In this article, we will go through some basic examples of Edge.js and see how Edge.js can be used to interact with SQL Server using Node.js.

Note: *This article assumes that you are aware of basics of Node.js. If you are new to it, please read the articles based on Node.js basics on dotnetcurry site.*

Getting Familiar with Edge.js

To bring .NET and Node.js together, Edge.js has some pre-requisites. It runs on .NET 4.5, so you must have .NET 4.5 installed. As Node.js treats all I/O and Network calls as slower operations, Edge.js assumes that the .NET routine to be called is a slower operation and handles it asynchronously. The .NET function

to be called has to be an asynchronous function as well.

The function is assigned to a delegate of type `Func<object, Task<object>>`. This means, the function is an asynchronous one that can take any type of argument and return any type of value. Edge.js takes care of converting the data from .NET type to JSON type and vice-versa. Because of this process of marshalling and unmarshalling, the .NET objects should not have circular references. Presence of circular references may lead to infinite loops while converting the data from one form to the other.

Hello World using Edge

Edge.js can be added to a Node.js application through NPM. Following is the command to install the package and save it to package.json file:

```
> npm install edge --save
```

The edge object can be obtained in a Node.js file as:

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

The edge object can accept inline C# code, read code from a .cs or .csx file, and also execute the code from a compiled dll. We will see all of these approaches.

To start with, let's write a "Hello world" routine inline in C# and call it using edge. Following snippet defines the edge object with inline C# code:

```
var helloWorld = edge.func(function () {
  /*async(input) => {
    return "Hurray! Inline C# works with
edge.js!!!";
  }*/
});
```

The asynchronous and anonymous C# function passed in the above snippet is compiled dynamically before calling it. The inline code has to be passed as a multiline comment. The method *edge.func* returns a proxy function that internally calls the C# method. So the C# method is not called till now. Following snippet calls the proxy:

```
helloWorld(null, function(error, result)
{
  if (error) {
    console.log("Error occured.");
    console.log(error);
    return;
  }
  console.log(result);
});
```

In the above snippet, we are passing a null value to first parameter of the proxy as we are not using the input value. The callback function is similar to any other callback function in Node.js accepting error and result as parameters.

We can rewrite the same Edge.js proxy creation by passing the C# code in the form of a string instead of a multiline comment. Following snippet shows this:

```
var helloWorld = edge.func(
  'async(input) => {'+
    'return "Hurray! Inline C# works
with edge.js!!!";'+
  '}'
);
```

We can pass a class in the snippet and call a method from the class as well. By convention, name of the class should be *Startup* and name of the method should be *Invoke*. The Invoke method will be attached to a delegate of type `Func<object, Task<object>>`. The following snippet shows usage of class:

It can be invoked the same way we did previously:

```
helloFromClass(10, function (error,
result) {
  if(error){
    console.log("error occured...");
    console.log(error);
    return;
  }
  console.log(result);

});
```

A separate C# file

Though it is possible to write the C# code inline, being developers, we always want to keep the code

in a separate file for better organization of the code. By convention, this file should have a class called Startup with the method *Invoke*. The Invoke method will be added to the delegate of type `Func<object, Task<object>>`.

Following snippet shows content in a separate file, Startup.cs:

```
using System.Threading.Tasks;

public class Startup {
  public async Task<object>
  Invoke(object input) {
    return new Person(){
      Name="Alex",
      Occupation="Software
      Professional",
      Salary=10000,
      City="Tokyo"
    };
  }
}

public class Person{
  public string Name { get; set; }
  public string Occupation { get; set; }
  public double Salary { get; set; }
  public string City { get; set; }
}
```

Performing CRUD Operations on SQL Server

Now that you have a basic idea of how Edge.js works, let's build a simple application that performs CRUD operations on a SQL Server database using Entity Framework and call this functionality from Node.js. As we will have a considerable amount of code to setup Entity Framework and perform CRUD operations in C#, let's create a class library and consume it using Edge.js.

Creating Database and Class Library

As a first step, create a new database named EmployeesDB and run the following commands to create the employees table and insert data into it:

```
CREATE TABLE Employees(
  Id INT IDENTITY PRIMARY KEY,
  Name VARCHAR(50),
  Occupation VARCHAR(20),
  Salary INT,
  City VARCHAR(50)
);

INSERT INTO Employees VALUES
  ('Ravi', 'Software Engineer', 10000,
'Hyderabad'),
  ('Rakesh', 'Accountant', 8000,
'Bangalore'),
  ('Rashmi', 'Govt Official', 7000,
'Delhi');
```

Open Visual Studio, create a new class library project named EmployeesCRUD and add a new Entity Data Model to the project pointing to the database created above. To make the process of consuming the dll in Edge.js easier, let's assign the connection string inline in the constructor of the context class. Following is the constructor of context class that I have in my class library:

```
public EmployeesModel()
  : base("data source=.;initial
catalog=EmployeesDB;
integrated security=True;
MultipleActiveResultSets=True;
App=EntityFramework;") {}
```

Add a new class to the project and name it EmployeesOperations.cs. This file will contain the methods to interact with Entity Framework and perform CRUD operations on the table Employees. As a best practice, let's implement the interface *IDisposable* in this class and dispose the context object in the *Dispose* method. Following is the basic setup in this class:

```
public class EmployeesOperations :
IDisposable {
  EmployeesModel context;

  public EmployeesOperations() {
    context = new EmployeesModel();
  }
  public void Dispose() {
    context.Dispose();
  }
}
```

As we will be calling methods of this class directly

using Edge.js, the methods have to follow signature of the delegate that we discussed earlier. Following is the method that gets all employees:

```
public async Task<object>
GetEmployees(object input)
{
  return await context.Employees.
ToListAsync();
}
```

There is a challenge with the methods performing add and edit operations, as we need to convert the input data from object to *Employee* type. This conversion is not straight forward, as the object passed into the .NET function is a *dynamic expando object*. We need to convert the object into a dictionary object and then read the values using property names as *keys*. Following method performs this conversion before inserting data into the database:

```
public async Task<object>
AddEmployee(object emp) {
  var empAsDictionary =
  (IDictionary<string, object>)emp;
  var employeeToAdd = new Employee() {
    Name = (string)
    empAsDictionary["Name"],
    City = (string)
    empAsDictionary["City"],
    Occupation = (string)
    empAsDictionary["Occupation"],
    Salary = (int)
    empAsDictionary["Salary"]
  };

  var addedEmployee = context.Employees.
  Add(employeeToAdd);

  await context.SaveChangesAsync();

  return addedEmployee;
}
```

The same rule applies to the edit method as well. It is shown below:

```
public async Task<object>
EditEmployee(object input) {
  var empAsDictionary =
  (IDictionary<string, object>)input;
  var id = (int)empAsDictionary["Id"];

  var employeeEntry = context.Employees.
SingleOrDefault(e => e.Id == id);
  employeeEntry.Name = (string)
  empAsDictionary["Name"];
  employeeEntry.Occupation = (string)
  empAsDictionary["Occupation"];
  employeeEntry.Salary = (int)
  empAsDictionary["Salary"];
  employeeEntry.City = (string)
  empAsDictionary["City"];

  context.Entry(employeeEntry).State =
  System.Data.Entity.EntityState.Modified

  return await context.
                SaveChangesAsync();
}
```

We will compose REST APIs using Express.js and call the above functions inside them. Before that, we need to make the compiled dll of the above class library available to the Node.js application. We can do it by building the class library project and copying the result dlls into a folder in the Node.js application.

Creating Node.js Application

Create a new folder in your system and name it 'NodeEdgeSample'. Create a new folder 'dlls' inside it and copy the binaries of the class library project into this folder. You can open this folder using your favorite tool for Node.js. I generally use WebStorm and have started using Visual Studio Code these days.

Add package.json file to this project using "npm init" command (discussed in Understanding NPM article) and add the following dependencies to it:

```
"dependencies": {
  "body-parser": "^1.13.2",
  "edge": "^0.10.1",
  "express": "^4.13.1"
}
```

Run NPM install to get these packages installed in the project. Add a new file to the project and name it 'server.js'. This file will contain all of the Node.js code required for the application. First things first, let's get references to all the packages and add the required middlewares to the Express.js pipeline. Following snippet does this:

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

```
var app = express();

app.use('/', express.
static(require('path').join(__dirname,
'scripts')));

app.use(bodyParser.urlencoded({
extended: true }));
app.use(bodyParser.json());
```

Now, let's start adding the required Express REST APIs to the application. As already mentioned, the REST endpoints will interact with the compiled dll to achieve their functionality. The dll file can be referred using the *edge.func* function. If type and method are not specified, it defaults class name as *Startup* and method name as *Invoke*. Otherwise, we can override the class and method names using the properties in the object passed into *edge.func*.

Following is the REST API that returns list of employees:

```
app.get('/api/employees', function
(request, response) {
 var getEmployeesProxy = edge.func({
  assemblyFile: 'dlls\\EmployeeCRUD.
  dll',
  typeName: 'EmployeeCRUD.
  EmployeesOperations',
  methodName: 'GetEmployees'
 });

 getEmployeesProxy(null,
 apiResponseHandler(request, response));
});
```

The function *apiResponseHandler* is a curried generic method for all the three REST APIs. This function returns another function that is called automatically once execution of the .NET function is completed. Following is the definition of this function:

```
function apiResponseHandler(request,
response) {
  return function(error, result) {
    if (error) {
      response.status(500).send({error:
      error});
      return;
    }
    response.send(result);
  };
}
```

Implementation of REST APIs for add and edit are similar to the one we just saw. The only difference is, they pass an input object to the proxy function.

```
app.post('/api/employees', function
(request, response) {
  var addEmployeeProxy = edge.func({
    assemblyFile:"dlls\\EmployeeCRUD.
    dll",
    typeName:"EmployeeCRUD.
    EmployeesOperations",
    methodName: "AddEmployee"
  });
  addEmployeeProxy(request.body,
  apiResponseHandler(request, response));
});

app.put('/api/employees/:id', function
(request, response) {
  var editEmployeeProxy = edge.func({
    assemblyFile:"dlls\\EmployeeCRUD.
    dll",
    typeName:"EmployeeCRUD.
    EmployeesOperations",
    methodName: "EditEmployee"
  });
  editEmployeeProxy(request.body,
  apiResponseHandler(request, response));
});
```

Consuming APIs on a Page

The final part of this tutorial is to consume these APIs on an HTML page. Add a new HTML page to the application and add bootstrap CSS and Angular.js to this file. This page will list all the employees and provide interfaces to add new employee and edit details of an existing employee. Following is the mark-up on the page:

```
<!doctype html>
<html>
<head>
 <title>Edge.js sample</title>
 <link rel="stylesheet" href="https://
 maxcdn.bootstrapcdn.com/bootstrap/
 3.3.5/css/bootstrap.min.css"/>
</head>
<body ng-app="edgeCrudApp">
 <div class="container" ng-
 controller="EdgeCrudController as vm">
  <div class="text-center">
   <h1>Node-Edge-.NET CRUD Application
   </h1>
   <hr/>
```

```
  <div class="col-md-12">
   <form name="vm.addEditEmployee">
    <div class="control-group">
     <input type="text" ng-
     model="vm.employee.Name"
     placeholder="Name" />
     <input type="text" ng-model=
     "vm.employee.Occupation"
     placeholder="Occupation" />
     <input type="text" ng-model="vm.
     employee.Salary" placeholder=
     "Salary" />
     <input type="text" ng-model="vm.
     employee.City" placeholder="City"
     />
     <input type="button" class="btn
     btn-primary" ng-click="vm.
     addOrEdit()" value="Add or Edit"
     />
     <input type="button" class="btn"
     value="Reset" ng-click="vm.
     reset()" />
    </div>
   </form>
  </div>
  <br/>
  <div class="col-md-10">
   <table class="table">
    <thead>
     <tr>
      <th style="text-align:
      center">Name</th>
      <th style="text-align:
      center">Occupation</th>
      <th style="text-align:
      center">Salary</th>
      <th style="text-align:
      center">City</th>
      <th style="text-align:
      center">Edit</th>
     </tr>
    </thead>
    <tbody>
    <tr ng-repeat="emp in
       vm.employees">
     <td>{{emp.Name}}</td>
     <td>{{emp.Occupation}}</td>
     <td>{{emp.Salary}}</td>
     <td>{{emp.City}}</td>
     <td>
      <button class="btn" ng-click="vm.
      edit(emp)">Edit</button>
     </td>
    </tr>
   </tbody>
  </table>
 </div>
</div>
</div>
```

```
<script src="https://ajax.googleapis.com/
ajax/libs/angularjs/1.4.3/angular.min.
js"></script>
<script src="app.js"></script>
</body>
</html>
```

Add a new folder to the application and name it 'scripts'. Add a new JavaScript file to this folder and name it 'app.js'. This file will contain the client side script of the application. Since we are building an Angular.js application, the file will have an Angular module with a controller and a service added to it. Functionality of the file includes:

- Getting list of employees on page load
- Adding an employee or, editing employee using the same form
- Resetting the form to pristine state once the employee is added or, edited

Here's the code for this file:

```
(function(){
 var app = angular.module('edgeCrudApp',
 []);
 app.controller('EdgeCrudController',
 function (edgeCrudSvc) {
  var vm = this;

  function getAllEmployees(){
   edgeCrudSvc.getEmployees().
   then(function (result) {
    vm.employees = result;
   }, function (error) {
      console.log(error);
   });
  }

  vm.addOrEdit = function () {
   vm.employee.Salary = parseInt(vm.
   employee.Salary);
   if(vm.employee.Id) {
    edgeCrudSvc.editEmployee(vm.
    employee)
     .then(function (result) {
      resetForm();
       getAllEmployees();
     }, function (error) {
      console.log("Error while updating
      an employee");
      console.log(error);
     });
    }
    else{
```

```
    edgeCrudSvc.addEmployee(vm.
    employee)
     .then(function (result) {
      resetForm();
      getAllEmployees();
    }, function (error) {
      console.log("Error while
      inserting new employee");
      console.log(error);
    });
  }
 };

 vm.reset= function () {
  resetForm();
 };

 function resetForm(){
  vm.employee = {};
  vm.addEditEmployee.$setPristine();
 }

 vm.edit = function(emp){
  vm.employee = emp;
 };

 getAllEmployees();
});

app.factory('edgeCrudSvc', function
($http) {
 var baseUrl = '/api/employees';

function getEmployees(){
 return $http.get(baseUrl)
 .then(function (result) {
  return result.data;
 }, function (error) {
   return error;
 });
}

function addEmployee(newEmployee){
 return $http.post(baseUrl,
 newEmployee)
  .then(function (result) {
   return result.data;
  }, function (error) {
    return error;
  });
 }

 function editEmployee(employee){
  return $http.put(baseUrl + '/' +
  employee.Id, employee)
   .then(function (result) {
    return result.data;
   }, function (error) {
     return error;
    });
   }

   return {
    getEmployees: getEmployees,
    addEmployee: addEmployee,
    editEmployee: editEmployee
   };
  });
}());
```

Save all the files and run the application. You should be able to add and edit employees. I am leaving the task of deleting employee as an assignment to the reader.

Conclusion

Generally it is challenging to make two different frameworks talk to each other. Edge.js takes away the pain of integrating two frameworks and provides an easier and cleaner way to take advantage of good features of .NET and Nodejs together to build great applications. It aligns with the Node.js event loop model and respects execution model of the platform as well. Let's thank Tomasz Jancjuk for his great work and use this tool effectively! ■

Download the entire source code from GitHub at
bit.ly/dncm20-edgenodedotnet

About the Author

Ravi Kiran *is a developer working on Microsoft Technologies. These days, he spends his time on JavaScript frameworks like AngularJS, ES6, ES7, Node.js, and Microsoft technologies like ASP.NET 5, C# and SignalR. He actively writes what he learns on his blog. He is rewarded with Microsoft MVP (ASP.NET/ IIS) and DZone MVB awards for his contribution to the community. You can follow him on twitter at @sravi_kiran*

DIAGNOSTIC ANALYZERS in Visual Studio 2015

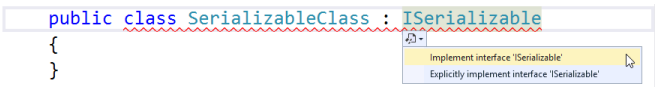
Visual Studio 2015 is finally delivering the results of project Roslyn – an ambitious attempt at rewriting the C# and Visual Basic compilers as a service. One of the benefits is the support for diagnostic analyzers in the code editor. They are small pieces of code for validating and refactoring different aspects of source code, which can easily be written by any developer wanting to improve his development experience in Visual Studio.

This article will help you learn everything you need to know about diagnostic analyzers, and hopefully convince you that for your everyday development, you need to start using them as soon as you switch to Visual Studio 2015.

You can download the Free Visual Studio 2015 Community Edition if you haven't already.

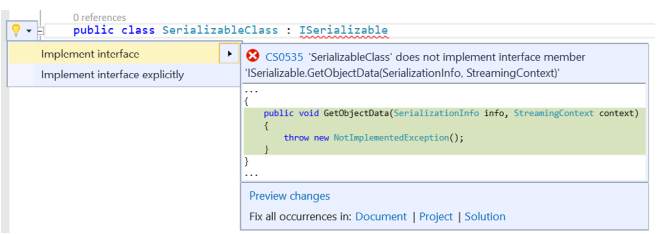
Improvements in the Code Editor

Visual Studio code editor has always performed background static code analysis to provide developers with information about errors and warnings in code. Instead of having to build the project and finding these errors and warnings in compiler output, they can be visualized directly in the source code, using squiggly lines below the offending code.



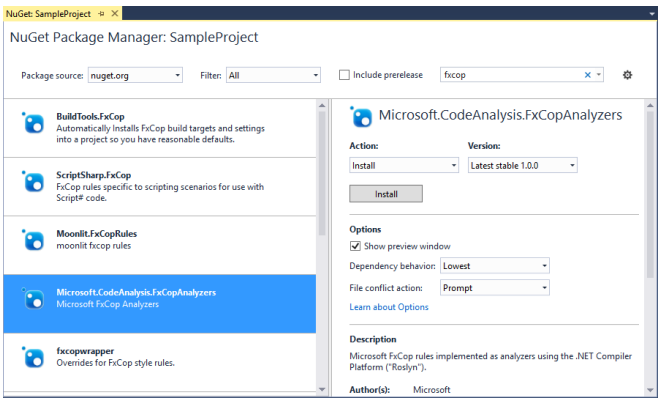
The functionality is still present in Visual Studio 2015, but it has been completely rewritten to take advantage of the new .NET Compiler Platform (formerly known as project Roslyn). Instead of the code editor parsing the code itself, it can now use the intermediate results of the actual C# (or Visual Basic) compiler, which is silently running in the background, compiling the code as the developer is changing it. The information obtained this way is more detailed and accurate, compared to what the code editor could produce.

Rewriting the Code editor introduced another improvement which is much easier to notice: *redesigned user interface*. This improvement is strongly influenced by JetBrains ReSharper and other third party Visual Studio extensions. All available actions for refactoring the affected code and fixing it, are now grouped together in a dropdown menu accessible via the lightbulb icon. Upon selecting the menu item for the selected refactoring, a preview is shown with all the changes, which is then applied to the code. This makes refactoring much safer and more inviting to use.



Adding Diagnostic Analyzers to a Project

With the new code editor, it is now possible to extend the built-in static code analysis on a per-project basis. To install a so-called diagnostic analyzer into a project, NuGet package manager is used. To open the NuGet window, right click on the project in *Solution Explorer* and select *Manage NuGet Packages*... from the context menu. Visual Studio 2015 includes NuGet 3, which opens a dockable window instead of a modal dialog, as was the case in the previous version.



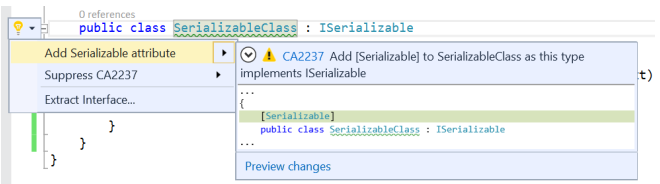
Unfortunately, at the time of this writing, there is no way to distinguish between packages containing diagnostic analyzers and ordinary libraries, which makes it impossible to browse through all currently available diagnostic analyzers. To test how they work, we will install a set of diagnostic analyzers, published by Microsoft: **Microsoft.CodeAnalysis.FxCopAnalyzers**. It is a reimplementation of some of the rules that were previously only available in a standalone tool, named **FxCop**, designed for checking conformance with the .NET framework design guidelines. To install the package into the current project, search for it in the official package source, select it in the list view, and click Install. After confirming two NuGet dialogs, the analyzers will get installed.

External Diagnostic Analyzers in Action

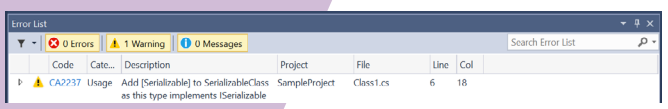
To see them in action, create a new class in your project containing the following code:

```
using System;
using System.Runtime.Serialization;

namespace ClassLibrary3 {
  public class SerializableClass :
  ISerializable {
    public void
    GetObjectData(SerializationInfo
    info, StreamingContext context) {
      throw new
      NotImplementedException();
    }
  }
}
```

Before the package was installed, there were no errors or warnings in this code. Immediately after the installation, `SerializableClass` became underlined with a green squiggly line indicating a warning. If you click on the lightbulb icon, you will notice a new suggestion for fixing the code.



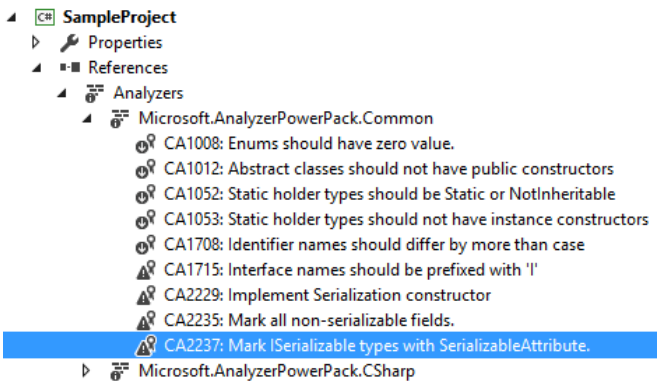
The behavior is the same as with built-in defects in all aspects, making it almost impossible to distinguish between them. The action is included in the same dropdown menu and it includes a preview of the change. There is even a hyperlink on the rule id (**CA2337**), which opens the corresponding help page, if you click on it. The warning is also displayed in Visual Studio's standard *Error List*. All of this is available to anyone developing a new diagnostic analyzer, not only to internal Microsoft developers.



Configuring the Behavior of Diagnostic Analyzers in the Project

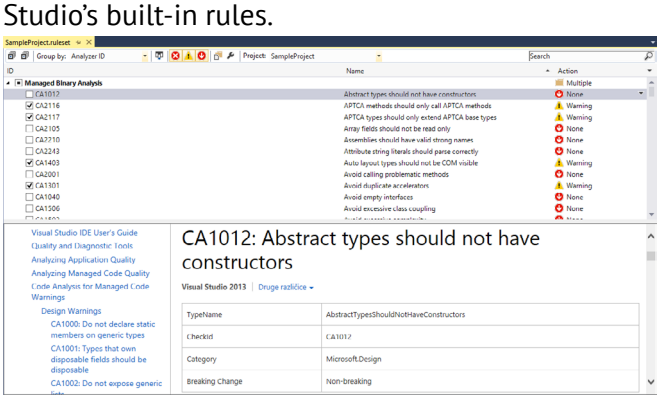
It is time to take a closer look at what happened when we installed the FxCop diagnostic analyzers into the project. As with all the other NuGet packages, the correct location is the project's *References* node in the *Solution Explorer* window.

Inside it is a new sub-node called *Analyzers*, which contains all the analyzers installed with our package. The rules are grouped by their assembly; each one of them having an icon in front, representing its severity: **Error**, **Warning**, **Info**, **Hidden**, or **None**.



When the package is first installed, all its rules have their default severity set. This can easily be changed for each individual project by right clicking on the rule item in the Solution Explorer window and checking the desired severity from the *Set Rule Set Severity* sub-menu of the context menu. Of course, this will also change the icon in front of that rule accordingly.

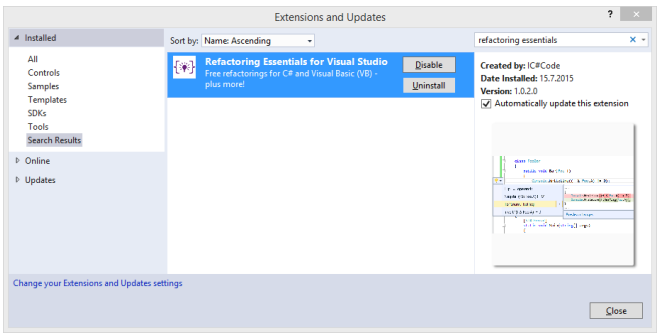
As soon as you change the severity of any rule, a new file is saved in the project folder: **<ProjectName>.ruleset**. Inside it, the severity for each of the enabled rules is stored. Once created, the file will also be added to the project root in the *Solution Explorer*. By double clicking it, you can open a dedicated editor for it, which makes it easier to enable/disable the rules and set their severity. For the selected rule, the editor also shows the online help page if it exists, providing a more detailed explanation which might be useful when the rule name is not clear enough. In contrast to the Analyzers node in the Solution Explorer window, the editor additionally allows configuration of Visual Studio's built-in rules.



In a collaborative environment, make sure you commit the ruleset file to source control along with the project file. The former contains rule severity settings, the latter a list of referenced analyzer assemblies. Both together will make sure that all developers, and the build server, will have the same configuration. As always, you should not put the **packages** folder into source control – the required assemblies will automatically be downloaded as part of the package restore operation when you attempt to build the project.

A Different Way of Installing Diagnostic Analyzers

Diagnostic analyzers are not necessarily distributed only as NuGet packages; they can also be available as Visual Studio extensions (VSIX files). These are listed in the **Visual Studio Gallery** and can be installed from the *Extensions and Updates* window in Visual Studio.



Analyzers and rules included from Visual Studio extensions will not be visible in the *Solution Explorer*; instead they will appear in the ruleset editor window. If you do not yet have a ruleset file in the project, it can be a bit tricky to open this window. You need to right click the empty **Analyzers** node in the Solution Explorer, and select the *Open Active Rule Set* from the context menu. This will open the default configuration, but as soon as you make your first change, the ruleset file will be created, just like when you first edited the rules in the *Solution Explorer* tree view.

Still, installing diagnostic analyzers as extensions does not work as well as when you are using NuGet packages. Although the rules are configured in the ruleset file, they are silently ignored when a developer does not have the same extension installed. This makes it impossible to strictly enforce the rule validation for the entire development team. In my opinion, diagnostic analyzers installed as extensions are only valuable during their development process, because this makes it easy to debug them from a second Visual Studio instance.

Current State of Diagnostic Analyzers Ecosystem

Since the final version of Visual Studio 2015 has just been released at the time of writing this article, it is still too early to expect a well-developed ecosystem of third party diagnostic analyzers. Nevertheless, some are already available, showing what we can expect in future.

The rules they are providing can be categorized into groups, based on the type of analysis they are doing:

- Validation of best practices for using APIs: e.g. gotchas to be aware of when developing for Microsoft Azure.

- Validation of other languages embedded as string literals: e.g. regular expression or SQL validation.

- Enforcing of design guidelines: most of FxCop rules belong here.

- Enforcing of style guidelines: naming policies, code formatting, preferred language features, etc.

Since there is no centralized directory of available diagnostic analyzers yet, it is difficult to know about all of them. Apart from the previously mentioned **Microsoft.CodeAnalysis.FxCopAnalyzers** and **Refactoring Essentials**, most of the other analyzers are less mature and they mostly seem to be created out of interest for learning or sample projects.

Conclusion

Visual Studio 2015 is a much larger step forward

in comparison to the previous versions – the main reason being the inclusion of Roslyn. Diagnostic analyzers might be the most important single addition to Visual Studio as a side product of that change.

I believe diagnostic analyzers can strongly affect the way we will be writing code, if they gain the traction they deserve.

In my opinion, the most valuable of them could be diagnostic analyzers for popular third party libraries, helping us write the code as it was meant to; assuming enough of them will be developed. The second important group will be rules for enforcing design and style guidelines. This field is currently covered with a couple of high profile Visual Studio extensions. Diagnostic analyzers from smaller developers could reinvigorate the innovation in this field.

The final important change could happen in large development teams with extensive internal codebase and strict coding standards. By configuring their rulesets and developing additional diagnostic analyzers for their own specific requirements, enforcing the rules that are already in place could become much more efficient and effective.

It will be interesting to see, where we end up in a year or two. In any case, do not wait that long before you try out diagnostic analyzers. It will be worth your time! ■

About the Author

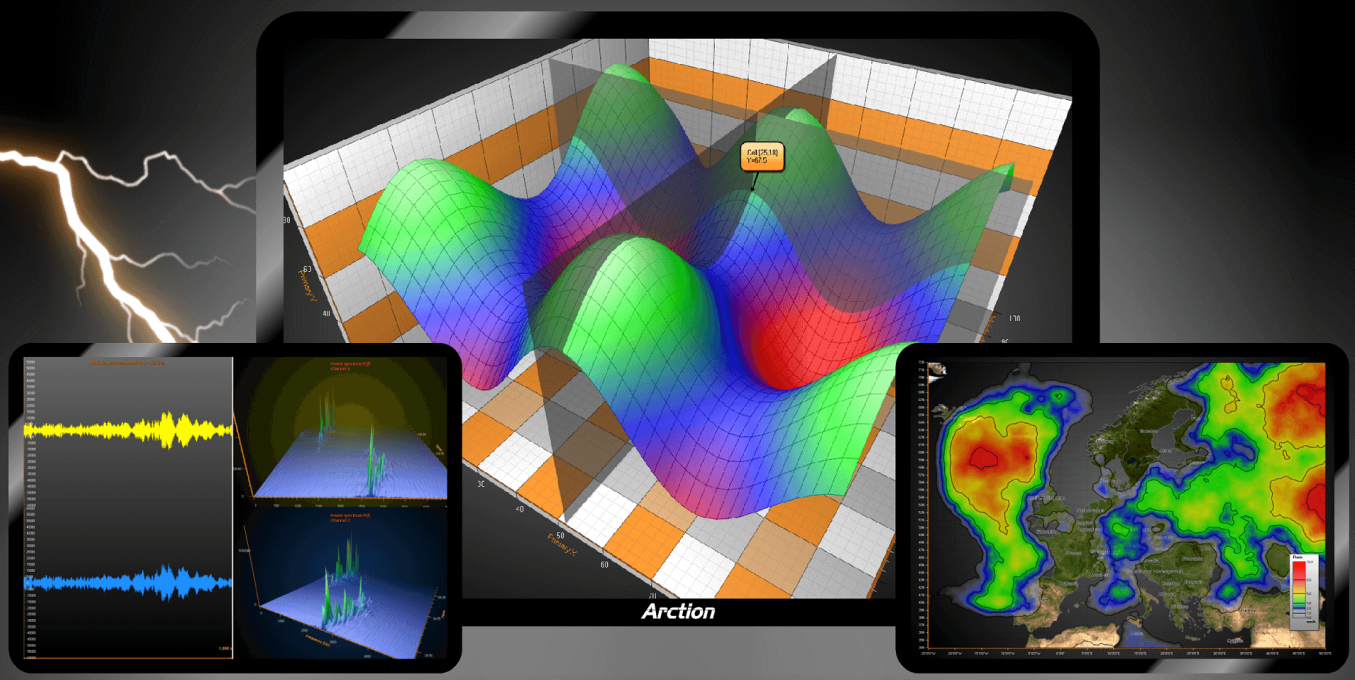
damir arh

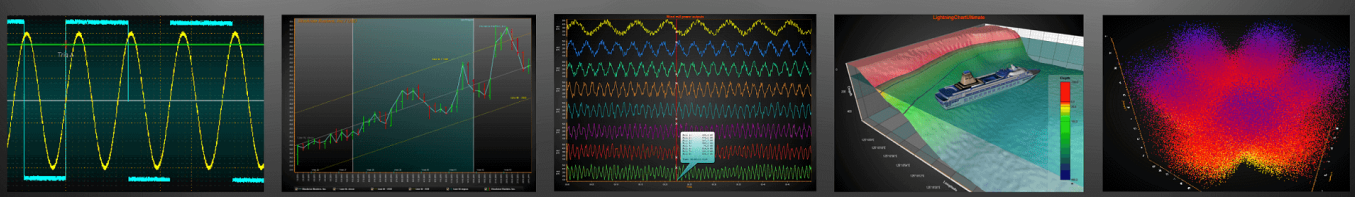
Microsoft® Most Valuable Professional

Damir Arh *has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.*

The FASTEST rendering Data Visualization components for WPF and WinForms...

LightningChart





HEAVY-DUTY DATA VISUALIZATION TOOLS FOR SCIENCE, ENGINEERING AND TRADING

WPF charts performance comparison

Opening large dataset	LightningChart is up to 977,000 % faster
Real-time monitoring	LightningChart is up to 2,700,000 % faster

WinForms charts performance comparison

Opening large dataset	LightningChart is up to 37,000 % faster
Real-time monitoring	LightningChart is up to 2,300,000 % faster

Results compared to average of other chart controls. See details at www.LightningChart.com/benchmark. LightningChart results apply for Ultimate edition.

- Entirely DirectX GPU accelerated
- Superior 2D and 3D rendering performance
- Optimized for real-time data monitoring
- Touch-enabled operations
- Supports gigantic data sets
- On-line and off-line maps
- Great customer support
- Compatible with Visual Studio 2005...2015
- Hundreds of examples

Download a free 30-day evaluation from

www.LightningChart.com

Arction

DevOps on Azure

*The concept of DevOps has been explained by many people in different ways. My interpretation of DevOps is '**Bringing Agility to Operations, in line with Development**'. I am going to start with two concepts in that definition that need some explanation and then move on to provide some examples of how Azure supports DevOps.*

The first concept in the statement I made about DevOps is ***Agility in Development***. Although it is not a new concept, it still needs explaining to many developers. Agility in general is how quick an entity reacts in a positive way to some change in the environment. One apt example of high level of agility is the fighter aircrafts engaged in a dogfight.

The aircraft and the pilot ecosystem has to take constant evasive and attacking actions, in response to similar actions taken by the enemy aircraft. In this example, one has to assume that the aircraft is built to take evasive actions required in the dogfight. If one imagines a passenger or bomber aircraft in place of the fighter aircraft, they will not be able to take those evasive actions. Extending this example to the teams that are developing software development, the teams that can take quick actions on changes detected in environment, like change in a business situation or rules of the business; is an Agile team. Like a passenger aircraft cannot be agile, not every team is Agile. Teams have to change or be recreated as an Agile team. A slow lumbering team cannot become Agile just like that, by following some Agile practices.

The second concept in that statement is ***Operations (team)*** and how it interacts with development team. Operations team has the responsibility of:

1. Creating the environment in which the developed application will run and provision it so that it can be used as and when required to deploy the developed application.

2. Ensure the quality of services provided to the customer like availability, performance, scalability, security etc.

3. When maintaining the quality of services requires some action from development team; operations team should provide the necessary feedback for that action.

4. Manage incidents related to the deployed software.

Development and Operations teams are two wheels of the chariot. If one is slow, the entire chariot does not become Agile. If the Development team does rapid increments to a product but if the Operations team takes long to deploy these increments, the benefits of agility shown by Development team cannot be accrued to the customer. In this article, I will show with examples how Azure can help Operations team also to become Agile.

Let us first take a simple example. A case where your team is given a task of creating a POC of a web application so that customer can view that POC and give budgetary sanctions for full-fledged application. You have decided that you will use Visual Studio to develop that POC, Visual Studio Online to do the source control with build and you will use Azure service to create a web application.

Let us start by creating that web application say called as "SSGSEMSWebApp". This application will have the domain of azurewebsites.net which is OK for us since it is only a POC. Once the application is provisioned, we will configure it to be deployed from source control by integrating it with VSO source control.

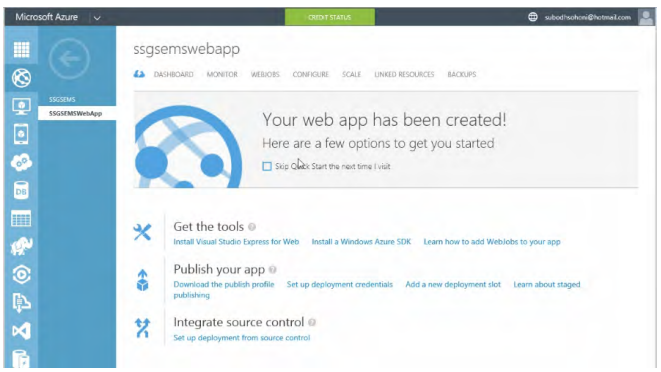


Figure 1: Create a Web Application on Azure

One assumption here is that we have already created a team project on VSO to integrate with this Azure web application. Right now there is nothing under the source control of that team project but we can link that with our web application. The name of that team project is say "SSGS EMS Demo".

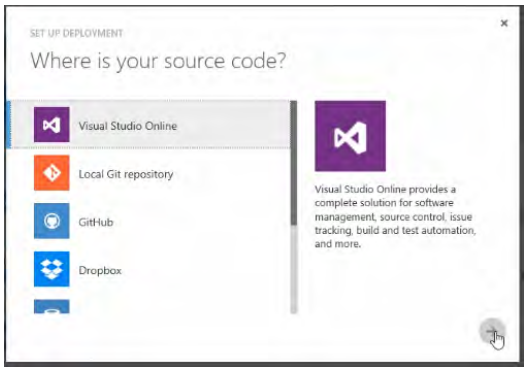


Figure 2: Connect Web App with VSO for Source Control

While adding this source control link, we will have to authorize it by logging on to our VSO subscription. Finally we select the team project name from dropdown as the repository to deploy.

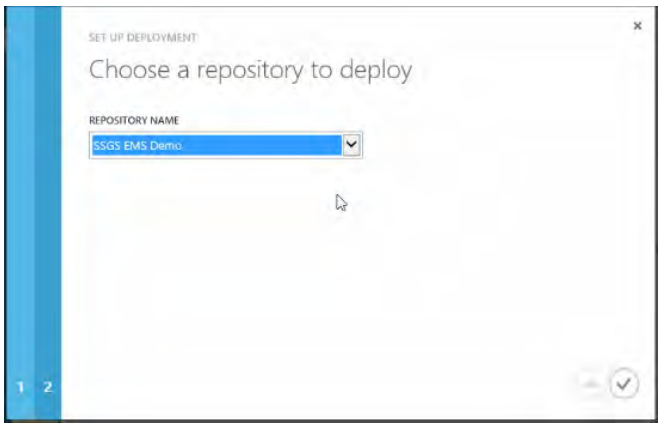


Figure 3: Select Team Project on VSO

Now we can open Visual Studio 2013 and create the actual ASP.NET web application. While creating it, we can chose to put it under the source control of SSGS EMS Demo team project as well as to host it in the cloud as a website. We can give the name of the application that we have created on Azure.

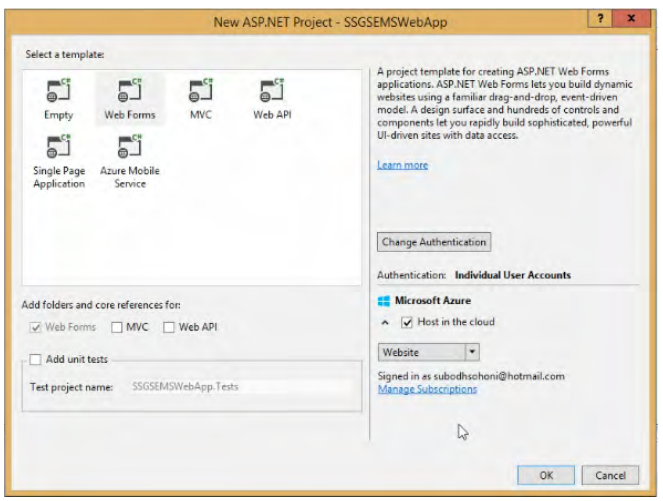


Figure 4: Create ASP.NET Application in Visual Studio

Figure 5: Add ASP.NET Application to Source Control

Once the application is ready in Visual Studio, we can do a check-in. You will observe that a build automatically starts as part of the Continuous Integration. This build definition is automatically created when we associated our Azure web application with the source control of this project.

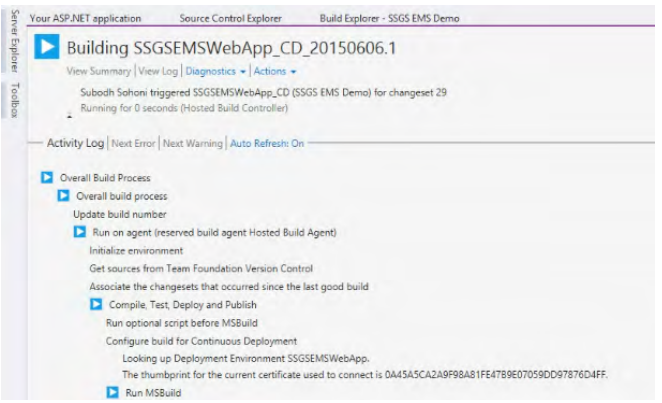


Figure 6: Automated Build with Check-in

One more interesting thing is that as part of the continuous integration, it also starts the deployment process as soon as the build is successfully over. It means that we get CICD – Continuous Integration Continuous Deployment as a bonus when we use Azure as the hosting provider and VSO as the source control of our application.

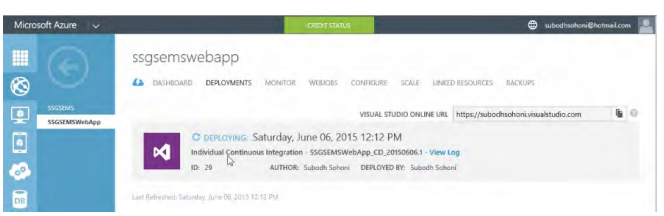


Figure 7: Continuous Deployment on Azure

This page will keep on showing the deployments that happen as part of CICD. We can even swap the deployments - if a new version is not working as expected, we can get the old version to be deployed with a single click of the mouse.

Operations team not only deploys but also monitors the application for quality of service. This is made simpler by a service that is provided in Azure. Name of that service is *Application Insights*. It collects telemetry data like availability, performance, errors etc. and allows us to view reports of that for applications hosted anywhere. They may be hosted in Azure or on a web server on premise or even running in Visual Studio 2013 for the data to be collected by Application Insights. For this data to be collected, the ASP.NET web application needs to have the App Insights SDK installed. It is a Nuget package that can be installed while creating the project, or even afterwards from the Solution Explorer of Visual Studio.

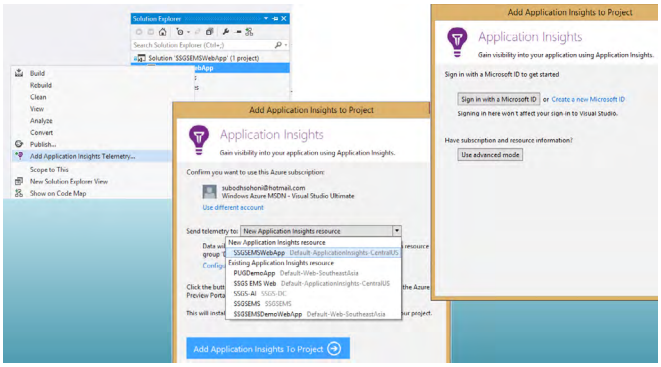


Figure 8: Setting Up Application Insights

Once the application starts running, telemetry data is collected and we can view it in the App Insights page of the new Azure portal.

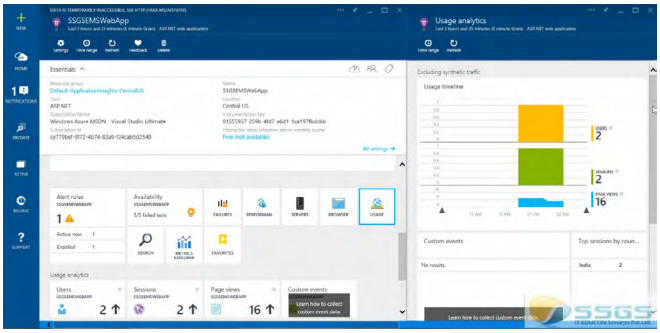


Figure 9: Results of Application Insights

It shows data related to availability, failures, overall performance, server performance, client side performance, usage patterns, User information, session information, page views and many others. We can drill down into each section to view details of the data.

Now that POC is shown to stakeholders and telemetry data has justified the application to be fully created, there are some constraints that are going to come. Let us list a typical set of constraints that operations team may encounter:

- Application will be hosted in actual VMs in Azure and not on the Web Application service of Azure

- There are stages where application will be tested in test environment, then in the preproduction environment UATs will be carried out and finally it will be deployed on the production environment.

- It will now be source controlled on on-premises TFS and built there only

- The environments that are required for hosting are in Azure. Each environment may contain multiple VMs.

- There has to be an approval from competent authority for promotion of the build from one environment to next.

In this case, it becomes appropriate to use *Microsoft Release Management 2013*. It can work with TFS or VSO. It can deploy software on machines on-premises or VMs in Azure. It can create and use multiple environments or multiple machines and those environments can be used in multiple stages that are part of the release path. Microsoft Release Management 2013 has all the necessary tools for deployment. It can do simple XCOPY deployment and can create web application with its own app pool. It can use DACPAC to do the deployment of SQL Server database and can also call PowerShell Desired State Configuration (DSC) scripts to do its tasks on a target machine. Let us now see how we can configure that.

First we have to create picklist of stage names that we will be using. For example: Testing, Pre-Prod and Production. Now we will create environments

with similar names. For that we have to provide the subscription details of Azure since we are creating the environments that are made up of Virtual Machines in Azure. One thing to remember is that all VMs that are to be part of same environment, have to be in the same Cloud Service. In fact, when we select Azure as the target for environments, the wizard shows the names of the Cloud Services in our subscription as possible candidates to create environments.

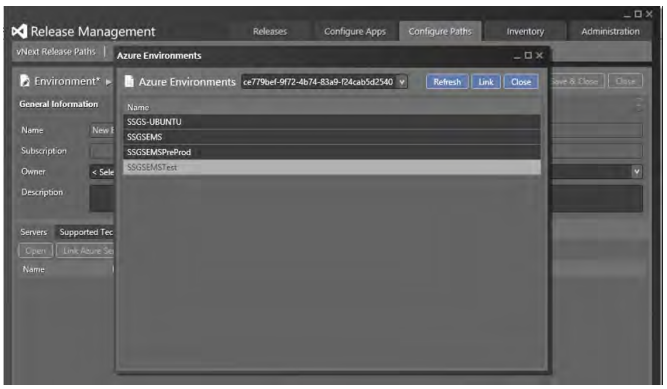


Figure 10: Select Cloud Service as Environments for Deployment

Now we can create the release path that depicts the stages in the serial order in which the deployment has to take place.

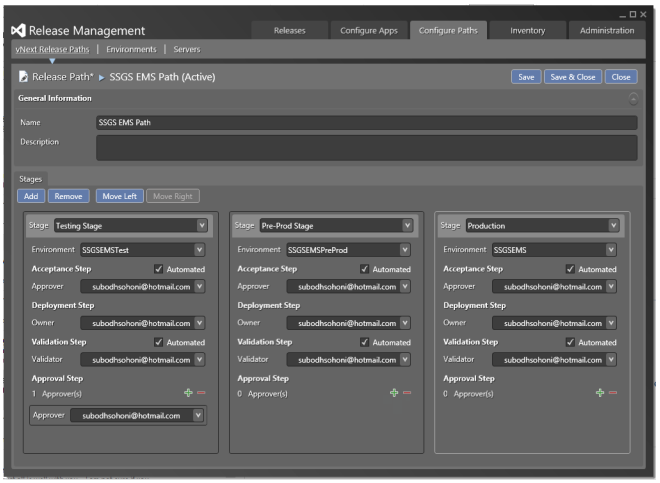


Figure 11: Define Release Path

You may have observed that for each stage there is a provision to add an approval step and give an approver name. This ensures that deployment on the next stage will happen automatically only when the approver has given an approval.

Final step is to create a release template where for each stage, we can configure which deployment actions have to take place and which tools are to be used for those actions. For Azure VMs, we are going to use PowerShell DSC script. We have to keep that as part of the project and then provide necessary details to the deployment step.

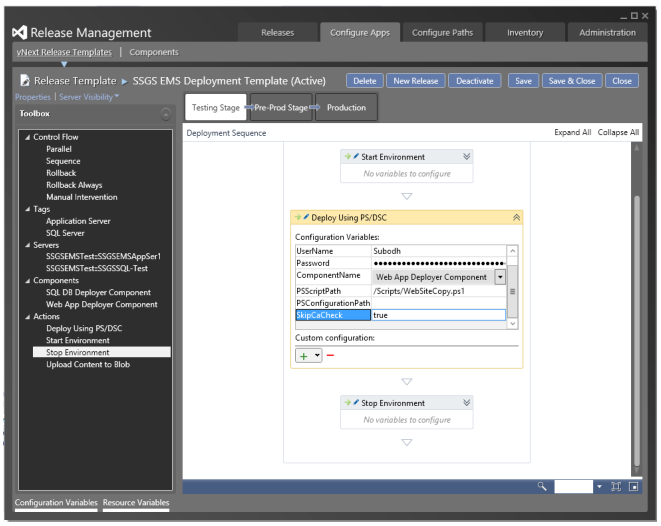


Figure 12: Define Deployment Actions

Once the template is ready, we can now either trigger the release manually or we can configure CICD with the help of build that is done using a custom build template that gets installed when Release Management is installed on TFS.

Deployment in a Mixed Environment (Linux and Windows VM's)

Next case that we are going to study is where deployment has to take place in a mixed environment containing some Linux VMs in addition to a Windows VM. Azure supports many templates of Linux to create VMs from them. What we want to do as part of deployment scenario, is the following:

1. Create an image of basic Linux VM

2. Provision it with some basic software such as Apache, MySQL and PHP. After provisioning we would like to store that VM for use and when required.

3. Once our application is ready, we would like to recreate the instance of that Linux VM

4. Finally, deploy the appropriate component of the application on it.

We can achieve this with the help of some open source tools that are supported on Azure. The first task of creating a basic Linux VM and then converting it into an image can be done using Azure Command Line Interface – CLI. Azure CLI is a set of open source commands that can do almost all tasks that you can execute from Azure Portal. We can use it to create a VM from standard Linux template and then store it as an image in the Azure Store.

Provisioning of VMs in Azure can be done using other open source tools. Two of the most popular tools in that are – Chef and Puppet. Both of these tools have similar capabilities. The concept behind working of these tools is *Desired State*. Each object on the VMs is treated as a resource and can have a desired state. Machine, OS, Users, Groups, Software, Services et al are treated as resources. From basic state to desired state, migration is achieved using scripts and inbuilt abilities of these tools.

Chef expects the desired state of the VMs to be described in scripts written in RUBY. These scripts are stored at a centralized location which by default is Chef Service that is available online. You may also optionally download and install Chef server on premises. This service has a management console that allows us to create the organization and download a starter kit containing RUBY script to start with.

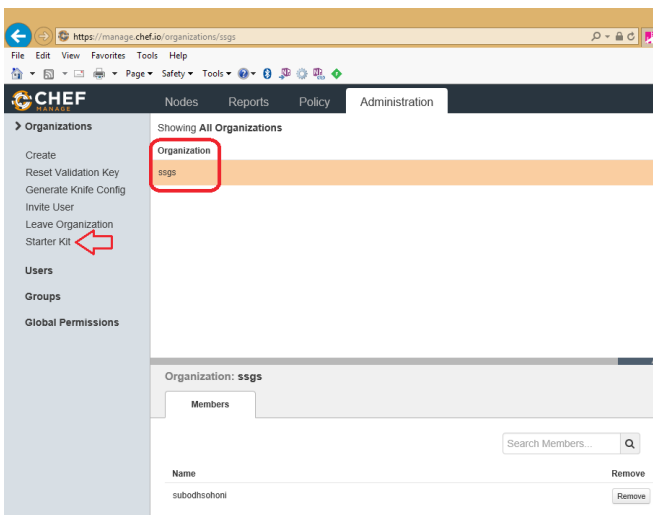


Figure 13: Setting Up Chef

We can extend these scripts to describe the desired state of the VMs. In the screenshot you may see that the *knife* tool is using the .publishsettings file of my subscription to connect to it and access VM under that.

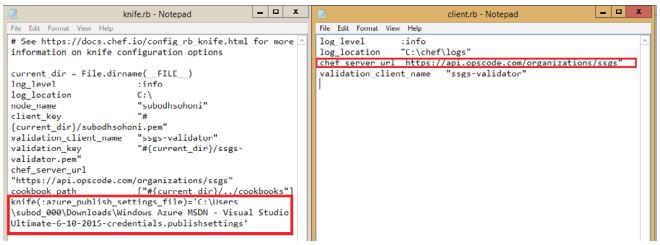


Figure 14: Chef Scripts in Ruby

On the virtual machine, we need the Chef Extension installed. Azure helps us install that on a VM that is being created.

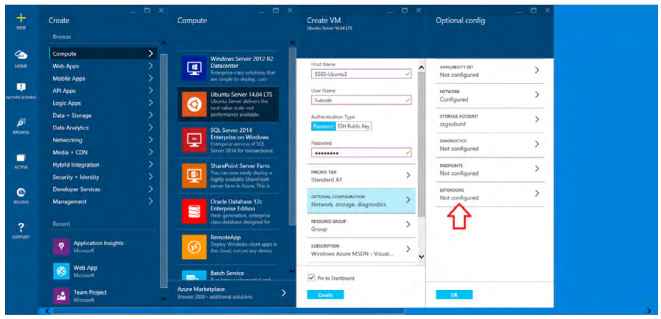


Figure 15: Add Chef Extension to VM on Azure

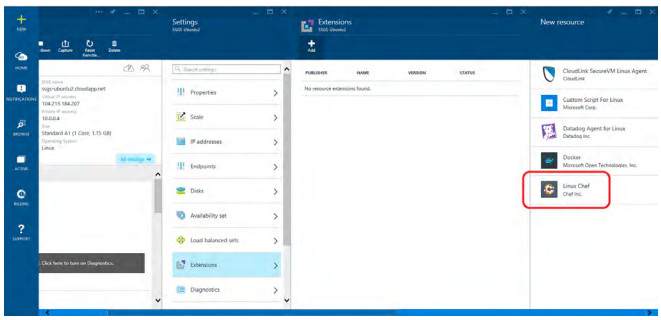


Figure 16: Select Linux Chef Extension

It now asks for client.rb file that is created earlier.

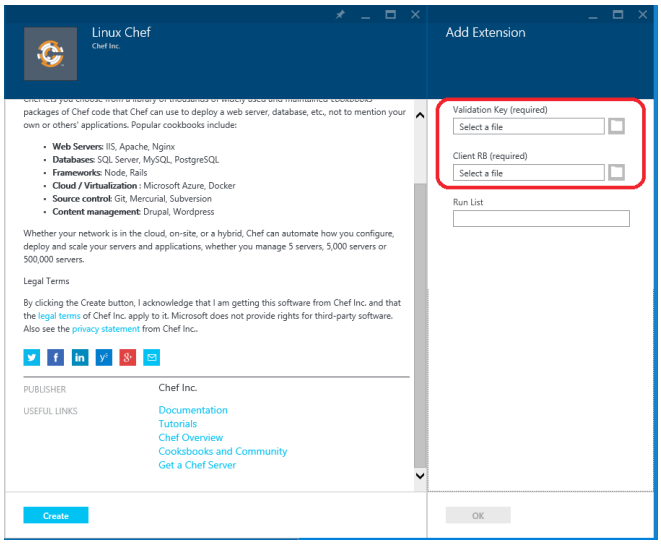


Figure 17: Provide Chef Scripts

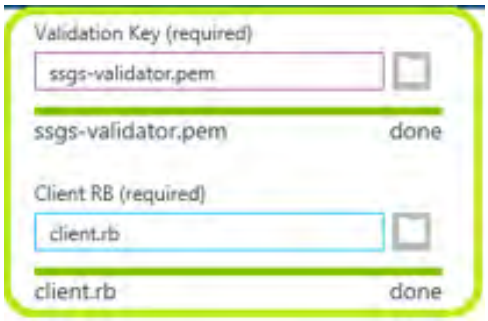


Figure 18: Uploaded Chef Scripts

The extension once installed will query the management service to get the DSC and implement it accordingly on the VM.

Puppet works on a similar concept but it has a server part of the software that has to be installed on one of our machines. It is called the *Puppet Master*. Azure provides a VM template for Puppet Master.

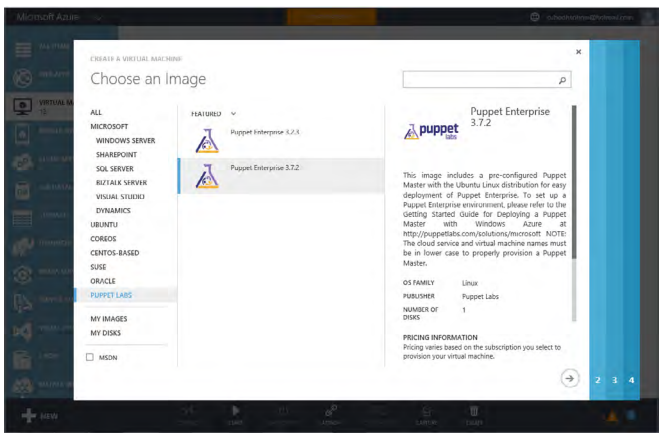


Figure 19: Puppet Master VM Template on Azure

Azure also provides the Puppet Agent to be installed on the VMs to be provisioned. We may also install the add-in provided by Puppet Labs to start creating the configuration scripts in Visual Studio.

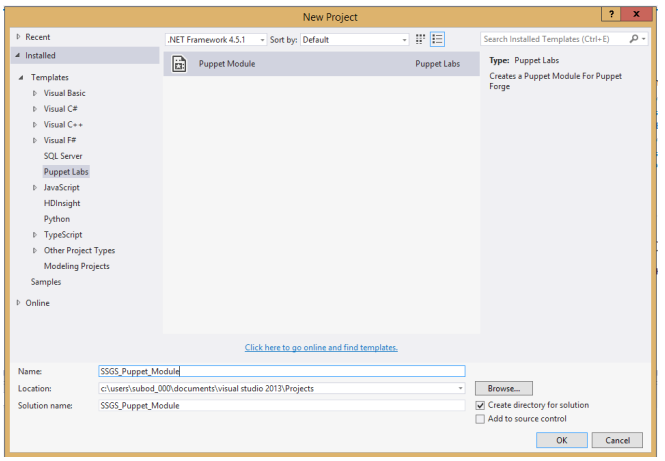


Figure 20: Puppet Scripting Project in Visual Studio

```
init.pp*  service_example.pp*
  # == Class: registry
  #
  # This class exists to prevent `include registry` from blowing up.
  #
  class registry {


  }

  class ssgsfile {
      file { "C:/inetpub/wwwroot/MySite":
          ensure => "directory",
          owner  => "Administrator",
          group  => "Administrators",
          mode   => 770,
      }
  }
```

Figure 21: Puppet Script

Our script can be put on the Puppet Forge (a shared resource repository) using the same add-in in Visual Studio, and then can be used from a *ssh* command prompt of target machine.

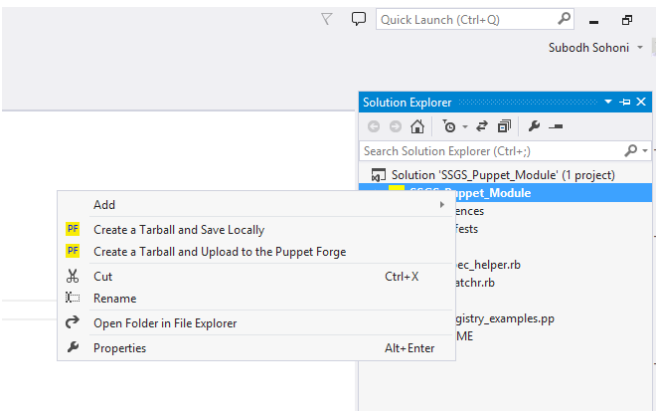


Figure 22: Deploying Puppet Script to Puppet Forge

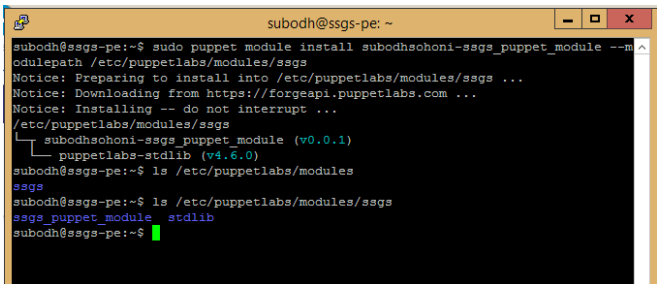


Figure 23: Execution of Puppet Script on Linux VM on Azure

Both Chef and Puppet can provision existing and running VMs. They cannot create a VM from an existing image. That can be done by another open source tool named *Vagrant*. It uses an image that is converted from an existing VM and then applies the configuration that is provided in a text file named *VagrantFile*. Microsoft Openness has recently created a provider for Azure so Vagrant can target creating VMs on Azure.

Summary

In this article we have seen various ways in which Azure supports the tasks of operations team as part of DevOps. By doing the automation of Continuous Integration Continuous Delivery (CICD), by creating the VMs and provisioning them, by automating the deployment of various components of your software on different machines, the agility of operations team that uses Azure services, can be increased to bring it in line with the development team ■

• • • • • • •

About the Authors

subodh sohoni

Microsoft® MVP Most Valuable Professional

***Subodh Sohoni**, Team System MVP, is an MCTS – Microsoft Team Foundation Server – Configuration and Development and also is a Microsoft Certified Trainer(MCT) since 2004. Subodh has his own company and conducts a lot of corporate trainings. He is an M.Tech. in Aircraft Production from IIT Madras. He has over 20 years of experience working in sectors like Production, Marketing, Software development and now Software Training. Follow him on twitter @subodhsohoni*

WPF ItemsControl - Part 2

Introduction

A casual glance at WPF's `ItemsControl` may not elicit much excitement, but behind its modest façade lies a wealth of power and flexibility. Gaining a deep understanding of the `ItemsControl` is crucial to your efficacy as a WPF developer. Such an understanding will enable you to recognize and rapidly solve a whole class of UI problems that would otherwise have been debilitating. This two part article will help you obtain this understanding.

As an added bonus, the knowledge you garner here will be applicable to the wider XAML ecosystem. Windows Store, Windows Phone, and Silverlight platforms all include support for `ItemsControl`. There may be slight differences in feature sets or usage here and there, but for the most part your knowledge will be transferrable.

In the first part of this article we covered the fundamentals of the `ItemsControl`. We're now going to build on these foundations and explore more advanced topics, such as **grouping and custom layouts**.

Advanced Item Appearance Customization

An obvious visual problem with our UI so far is the lack of sufficient spacing around and between items. Of course, we could address this by adding margins to the `Grid` of each `DataTemplate`, but that would result in duplicate code[1] and would couple our templates more tightly to a specific context. If we want to re-use those templates elsewhere, the margin we choose here may not suit the new location.

A better way to solve this is by way of the `ItemContainerStyle` for the `ItemsControl`. As the name suggests, this property allows us to provide a `Style` that is applied to the container for each item. But what is the container for our items? For a vanilla `ItemsControl`, it is a simple `ContentPresenter`, which we can verify by using the WPF visualizer (see Figure 1) The container used can differ depending on the specific subclass of `ItemsControl` we're using, as we'll see later. For now, note that the `Style` we set for `ItemContainerStyle` will be applied to a `ContentPresenter`.

[1] OK, we could move the margin into a resource and share that resource, but that's still painful.

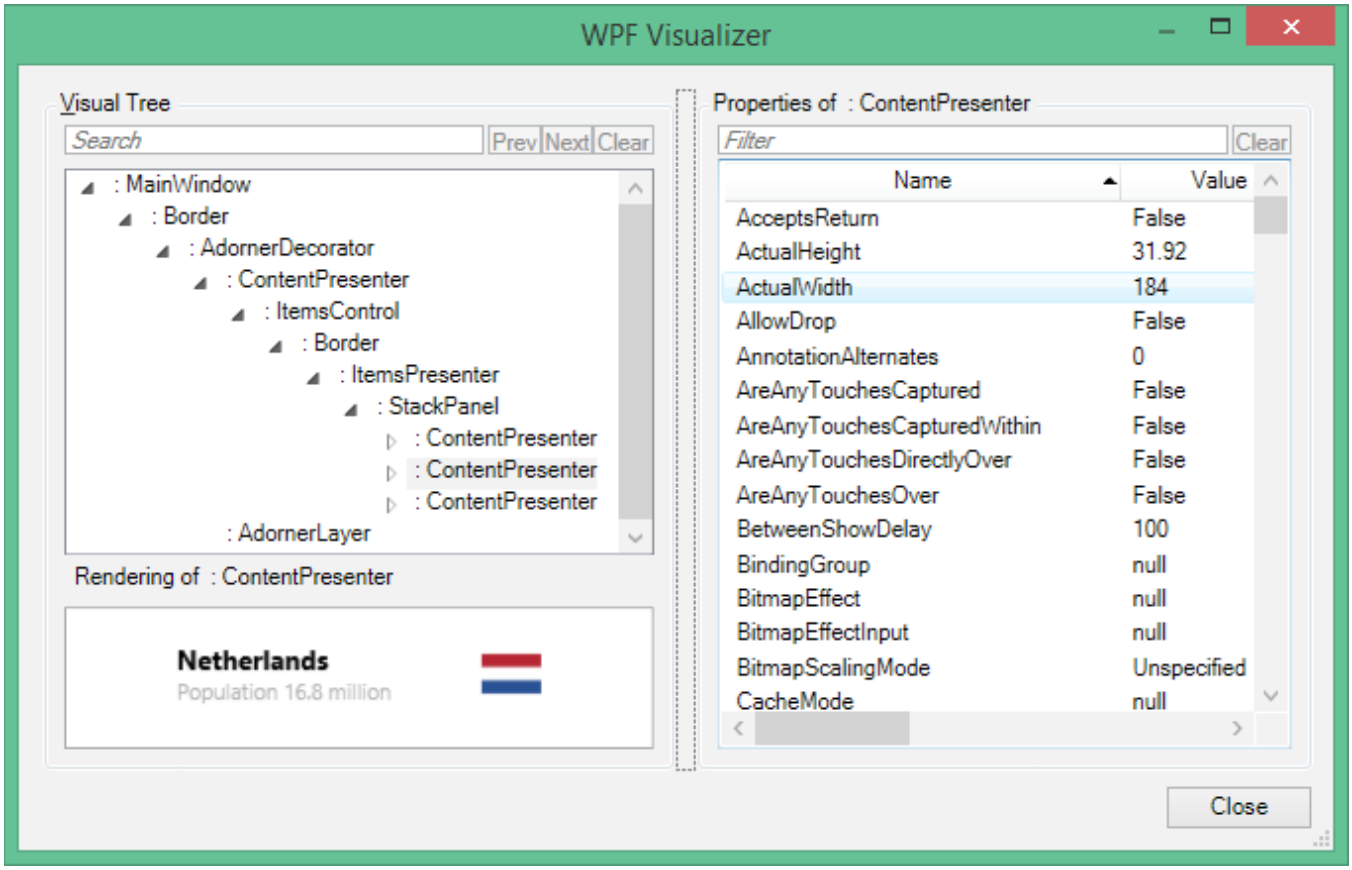


Figure 1: The container for items in a vanilla ItemsControl is a ContentPresenter

Thus, we can modify our XAML to specify a `Margin` for the `ContentPresenter` hosting each of our items:

```
<ItemsControl
   ItemsSource="{Binding}"
   ItemTemplateSelector="{StaticResource
   PlaceDataTemplateSelector}">
 <ItemsControl.ItemContainerStyle>
  <Style TargetType="ContentPresenter">
   <Setter Property="Margin" Value="6 3
   6 3"/>
  </Style>
 </ItemsControl.ItemContainerStyle>
</ItemsControl>
```

This gives us the more aesthetically pleasing UI depicted in Figure 2. Things aren't quite so cluttered anymore.

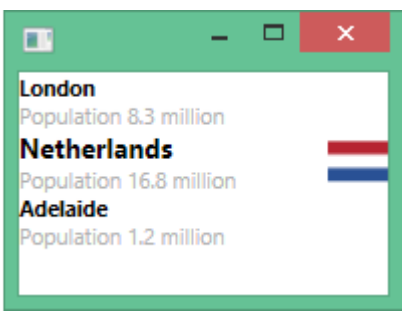


Figure 2: Using ItemContainerStyle to add some spacing around items

Just as the `DataTemplate` property has a corresponding `DataTemplateSelector` property, so too the `ItemContainerStyle` property has a corresponding `ItemContainerStyleSelector`. The mechanics of this property are exactly the same as `DataTemplateSelector`, so I won't be going into the details here. The only difference is you're returning a `Style` instead of a `DataTemplate`. In choosing the `Style`, you're privy to the same information as you are when choosing a `DataTemplate:` the item itself, and the container for the item.

When you have many items in a list, it's often helpful to visually distinguish one item from the next in some subtle fashion. Otherwise, in the eyes of the user, it can be hard to tell where one item ends and the next begins. Sufficient spacing is one way to achieve this, but when the volume of data is high and screen real estate is of high importance, an alternating background color from one item to the next is another common technique.

We've just discussed how we can dynamically choose a style by using `ItemContainerStyleSelector`. Would that be a

viable means of achieving alternating background colors? Not really. After all, how would we know which `Style` to choose? Our view model would need to expose its index within the list so that we could choose one `Style` for even indices and another for odd indices. And what happens when we add or rearrange items in our list? Suddenly we have to invalidate a whole bunch of items in order for the UI to pick up the correct styles per item. Obviously, such an approach would be awful.

Thankfully, `ItemsControl` provides the `AlternationCount` and `AlternationIndex` properties to help us out of this predicament. These two properties work in tandem: `AlternationCount` allows us to specify how many items to count before resetting `AlternationIndex` back to zero, and `AlternationIndex` (an attached property) tells us what index a certain item container has been assigned. For example, if we set `AlternationCount` to 2 then the first item will have `AlternationIndex` 0, the second item will have `AlternationIndex` 1, and the third item will have `AlternationIndex` 0 again. Repeat *ad nauseam* for all items in our list. The upshot of this is that our item containers now have an `AlternationIndex` that we can use to trigger different visuals.

Let's try this out. The first part is easy:

```
<ItemsControl
  ItemsSource="{Binding}"
  ItemTemplateSelector="{StaticResource
  PlaceDataTemplateSelector}"
  AlternationCount="2">
```

We've now specified an `AlternationCount` of 2, but how do we select a different background based on the resulting `AlternationIndex` of each container? The first step is to introduce a trigger into our `Style`:

```
<Style TargetType="ContentPresenter">
  <Setter Property="Margin" Value="6 3 6
  3"/>

  <Style.Triggers>
    <Trigger Property="ItemsControl.
    AlternationIndex" Value="1">
    </Trigger>
  </Style.Triggers>
</Style>
```

However, we now have a problem. `ContentPresenter` is too primitive a control to even expose a `Background` property. We'll come back to this issue shortly, but for now let's instead modify the item's `Margin` based on the `AlternationIndex`:

```
<Trigger Property="ItemsControl.
AlternationIndex" Value="1">
  <Setter Property="Margin" Value="12 3
  12 3"/>
</Trigger>
```

The result is shown in Figure 3 . The first thing you'll notice is that I've added some more data to make our changes easier to spot. Secondly, you'll see that every other item is indented further than the item preceding it (both from the left and the right). Success!

Figure 3: Using AlternationCount and AlternationIndex to affect item margins

Now, to the problem of wanting to set the `Background` rather than the `Margin`. Fundamentally, we need a different container for our items. There are ways of cobbling a solution together using `ContentPresenter`, but they're essentially just hacks. A better approach is to tell our `ItemsControl` to use a `ContentControl` to house each item – that way we can modify the `Background` of the `ContentControl`. It's worth pointing out that this is not usually necessary. It's relatively rare to use an `ItemsControl` directly as opposed to one of its derived classes. The derived classes all provide richer containers for their items, and those containers will generally facilitate background color changes. But we're going to get

to derived classes later, and this is an instructive exercise besides.

`ItemsControl` includes a property called `ItemContainerGenerator`, which looks like a likely candidate for the customizations we require. Alas, this property is get-only and `ItemContainerGenerator` is a sealed class. Not to worry though, because we can do what other `ItemsControl` derivative classes do: override a method called `GetContainerForItemOverride`:

```
public sealed class CustomItemsControl :
ItemsControl
{
  protected override DependencyObject
  GetContainerForItemOverride()
  {
    return new ContentControl();
  }
}
```

If we use our `CustomItemsControl` instead of the default `ItemsControl`, our items are now wrapped in a `ContentControl`, as per Figure 4. And you'll notice the property list contains our old friend, `Background`.

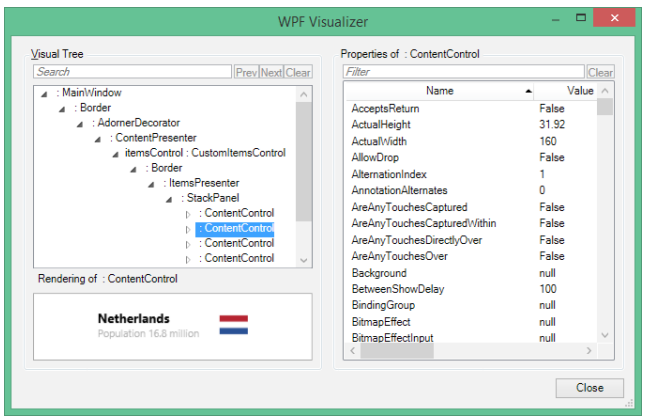


Figure 4: Our ItemsControl using ContentControl to wrap each item instead of ContentPresenter

Let's modify our `Style` then:

```
<Trigger Property="ItemsControl.
AlternationIndex" Value="1">
  <Setter Property="Background"
  Value="LightGray"/>
</Trigger>
```

Seems like all we'd need to do, right? Unfortunately, no. Even though `ContentControl` has a `Background` property, its default template does nothing with it. WPF is not making our lives easy right now! To rectify this, let's include our own template for `ContentControl`:

```
<local:CustomItemsControl.
ItemContainerStyle>
 <Style TargetType="ContentControl">
  <Setter Property="Padding" Value="6 3
  6 3"/>
  <Setter Property="Template">

   <Setter.Value>
    <ControlTemplate
    TargetType="ContentControl">
     <Border
      Background="{TemplateBinding
      Background}"
      Padding="{TemplateBinding
      Padding}">
      <ContentPresenter/>
      </Border>
     </ControlTemplate>
    </Setter.Value>
   </Setter>

   <Style.Triggers>
   <Trigger Property="ItemsControl.
   AlternationIndex" Value="1">
    <Setter Property="Background"
    Value="LightGray"/>
   </Trigger>
  </Style.Triggers>
 </Style>
</local:CustomItemsControl.
ItemContainerStyle>
```

You'll notice I've also set `Padding` on our `ContentControl` and changed our `Margin` assignment to instead modify `Padding`. This is to ensure that our background color shows right up to the edges of our `ItemsControl`, rather than being inset. The end result is shown in Figure 5.

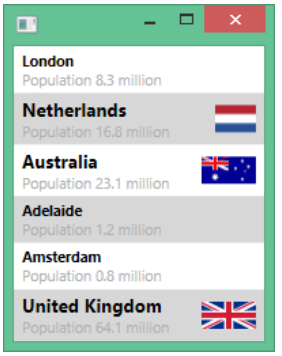


Figure 5: Our ItemsControl with alternating background colors for each item

Phew! That wasn't quite the smooth sailing we hoped for, but it's worth re-iterating that we're on a lesser travelled path here. Using other controls that subclass `ItemsControl` will usually result in these kinds of scenarios working more smoothly.

Grouping Items

When you bind an `ItemsControl` to a collection as we have been thus far, things are not entirely as they seem. Behind the scenes, WPF is creating and binding to an adapter called `CollectionViewSource` instead of binding directly to the collection we give it. The `CollectionViewSource` that is created on our behalf will point to the collection we supplied. This layer of indirection provides a suitable place where grouping, sorting, and filtering of the underlying data can occur. We're not going to be discussing sorting and filtering in this article because they're entirely data concerns, but let's take a look at how we can group our data.

The grouping functionality within the `CollectionViewSource` works in tandem with grouping properties on the `ItemsControl` itself. The `CollectionViewSource` decides how data is grouped whereas the `ItemsControl` decides how to render those groups.

Suppose we want to group our data by the type of place in question: all countries in one group, and all cities in another. The first thing we need to do is stop binding directly to our collection of places, and instead bind to a `CollectionViewSource`. There are various ways of achieving this, but we'll do it in XAML. As a first step, let's explicitly declare a `CollectionViewSource` in our `Window` resources:

```
<CollectionViewSource x:Key="places"
Source="{Binding}"/>
```

Notice how the `Source` property of the `CollectionViewSource` is bound to our underlying data collection. Now we need to modify our `ItemsSource` to use this `CollectionViewSource`. The syntax for doing so is a little tricky:

```
<local:CustomItemsControl
  ItemsSource="{Binding
  Source={StaticResource places}}"
  …
```

Notice that we're binding the `ItemsSource` property instead of simply assigning the `CollectionViewSource` directly to it. That's because it is WPF's binding infrastructure that recognizes the special role of `CollectionViewSource` and interacts with it to obtain our data. In other words, the binding infrastructure understands the indirection that `CollectionViewSource` provides, and it resolves this indirection on our behalf.

With those two changes in place, our UI looks exactly as per Figure 5. But now what happens if we introduce some grouping to our `CollectionViewSource`? To do so, we need to include one or more `GroupDescription` objects when creating the `CollectionViewSource`. Often these will be instances of `PropertyGroupDescription`. For example, if we were grouping by the `Name` property on our view models (which might make sense with a lot more data), we could do this:

```
<CollectionViewSource x:Key="places"
Source="{Binding}">

  <CollectionViewSource.
  GroupDescriptions>
    <PropertyGroupDescription
    PropertyName="Name"/>
  </CollectionViewSource.
  GroupDescriptions>
</CollectionViewSource>
```

Then items with the same name will appear together (that is, if we *had* any with the same name). But this approach doesn't help us in this scenario because we want to group by the type of place, and we have no appropriate property that gives us that information. Sure, we could add such a property, but we're going to instead demonstrate the flexibility of the grouping infrastructure by sub-classing the abstract `GroupDescription` class:

```
public sealed class
PlaceTypeGroupDescription :
GroupDescription
{
  public override object
  GroupNameFromItem(object item, int
```

```
  level, CultureInfo culture)
  {
    return item is CityViewModel ?
    "Cities" : "Countries";
  }
}
```

We have only the one method to implement, and it simply returns either "Cities" or "Countries" depending on the type of the item it's given. In other words, it dictates the group to which the item belongs to. Let's update our XAML to use our custom `GroupDescription`:

```
<CollectionViewSource x:Key="places"
Source="{Binding}">
  <CollectionViewSource.
  GroupDescriptions>

  <local:PlaceTypeGroupDescription/>
    </CollectionViewSource.
    GroupDescriptions>
</CollectionViewSource>
```

Now when we run the application we see Figure 6. It's an encouraging change: all our cities now appear together at the top, followed by all countries below them. But why is there nothing to visually demarcate each group? That's because we haven't supplied a `GroupStyle`.

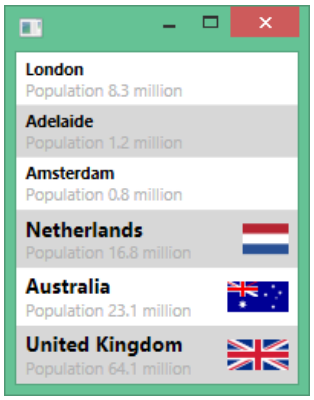


Figure 6: Our data grouped by place type

The `GroupStyle` property on `ItemsControl` allows us to control how groups are rendered. It's important to realize that the property's type is `GroupStyle`, not `Style`. The `GroupStyle` class defines a number of properties that we can use to control aspects of the group's appearance.

The `HeaderTemplate` (and `HeaderTemplateSelector`) properties allow us to control what is rendered for each group's header area, whereas the `ContainerStyle` and `ContainerStyleSelector` properties do the same for the container of each item within the group. We can also specify an `AlternationCount` like we did for `ItemsControl`, but this time each group is being assigned an `AlternationIndex` rather than the items within the group. The `Panel` property enables us to choose the layout panel that hosts the groups (we'll be covering layout customization in a later section).

This is all very empowering, but let's just try and display the name of each of our groups. We can do that by adding this within our `ItemsControl` definition:

```
<local:CustomItemsControl.GroupStyle>
 <GroupStyle>
  <GroupStyle.HeaderTemplate>
   <DataTemplate>
    <Border Padding="6"
    Background="DarkSlateGray">
     <TextBlock
      Text="{Binding Name}"
      FontSize="14pt"
      FontWeight="Bold"
      Foreground="White"/>
    </Border>
   </DataTemplate>
  </GroupStyle.HeaderTemplate>
 </GroupStyle>
</local:CustomItemsControl.GroupStyle>
```

This results in Figure 7. If you've got an eye for design, you're surely cringing right now. Sorry about that, but as you can see, we've successfully labelled each of our groups.

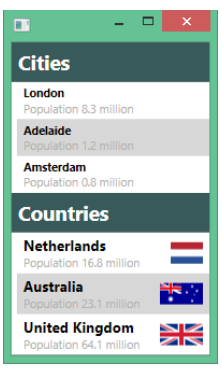


Figure 7: Headers for our groups

We can take things a lot further with `GroupStyle`. For example, by tweaking the `ContainerStyle` and `HeaderTemplate`, I was quickly able to come up with Figure 8. Of course, I chose a fresh palette and tweaked the font as well, but you get the idea.

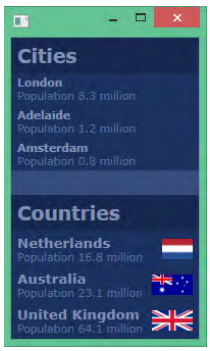


Figure 8: A visually tweaked version of our grouped data

`ItemsControl` also includes a `GroupStyleSelector` property, which offers us the now familiar flexibility to dynamically select a `GroupStyle` instead of hard-wiring one. And, as a final note, it has an `IsGrouping` property, which tells you whether data is grouped or not. This might be useful when authoring control templates because one can trigger different visuals according to whether data is being grouped or not.

Adjusting Item Layout

To this point, the items within our `ItemsControl` have always been stacked one above the other. That's certainly the most common way of laying out lists of items, but it's not the only way. And if you consider the fact that `ItemsControl` is designed to be used in a wide variety of scenarios, one begins to suspect there must be a way to influence the layout of items. And indeed there is.

By default, `ItemsControl` hosts its items within a `StackPanel` with a vertical orientation. If you were paying careful attention, you may have noticed this within the WPF Visualizer (see Figure 4, for example). The container for each item within the `ItemsControl` is added, in order, to this panel. That explains the vertical stacking we've seen up until now.

But the `ItemsPanel` property on `ItemsControl` allows us to provide our own panel in which items will be hosted. More accurately, it allows us to specify a template with which to produce the panel.

Suppose, for example, we want to lay out our places horizontally rather than vertically. To achieve this, we can write the following XAML:

```
<local:CustomItemsControl …>
 <local:CustomItemsControl.ItemsPanel>
  <ItemsPanelTemplate>
   <StackPanel Orientation=
   "Horizontal"/>
  </ItemsPanelTemplate>
 </local:CustomItemsControl.ItemsPanel>
 …
```

Things are a bit verbose here due to our custom `ItemsControl`. Without such customization, it would look more like this:

```
<ItemsControl …>
 <ItemsControl.ItemsPanel>
  <ItemsPanelTemplate>
   <StackPanel Orientation=
   "Horizontal"/>
  </ItemsPanelTemplate>
 </ItemsControl.ItemsPanel>
 …
```

In any case, what we end up with is depicted in Figure 9. Notice how our items now run left to right. In other words, they're stacked horizontally rather than vertically. You can also see that each group of items has its own panel. That explains why we supply a template for the panel rather than the panel itself: the `ItemsControl` needs to be able to dynamically create these panels at whim to accommodate the addition of new groups.

Figure 9: Our ItemsControl with a horizontally-oriented StackPanel

Of course, we don't even have to use a `StackPanel`. Any `Panel` will do, though some are less suited to situations where there could be any number of children added to them. What if we wanted two equally wide columns of data? We can do that with a `UniformGrid`:

```
<ItemsPanelTemplate>
  <UniformGrid Columns="2"/>
```

```
</ItemsPanelTemplate>
```

This gives us Figure 10. Or we could lay out items such that they automatically wrap when there is insufficient space remaining on the current row:

```
<ItemsPanelTemplate>
  <WrapPanel Orientation="Horizontal"/>
</ItemsPanelTemplate>
```

This gives us Figure 11. Notice how the items in our groups adapt to the available space. If they fit, they will be displayed left to right. If there is no more space left, they wrap to the next line. We can also control how *groups* are laid out by assigning an `ItemsPanelTemplate` instance to the `Panel` property on `GroupStyle`. The principals are exactly the same as when laying out items within the group, but the panel will instead lay out the groups themselves.

Figure 10: Laying out items in two columns by using a UniformGrid

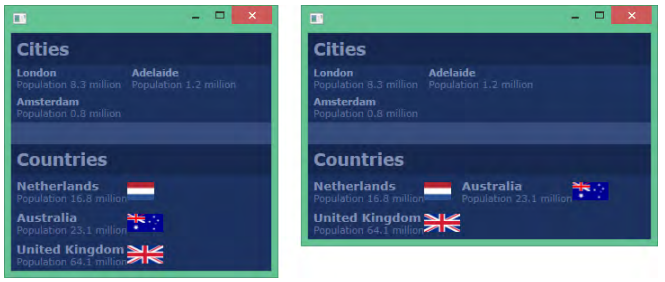


Figure 11: Wrapping items according to the available space

We can take this a lot further. What if we wanted to display each item on a world map? The first thing we'll need is the longitude and latitude for each place. To that end, we can simply add the appropriate properties to our `PlaceViewModel` and populate those properties using google as an aid. For example, here's the new constructor invocation for London:

```
new CityViewModel(
  "London",
  8.308f,
  -0.1275,
  51.5072),
```

Notice the final two arguments. They are London's longitude and latitude.

Next, we need to display a map. I found a free Mercator-projected SVG online and exported it to a PNG using Inkscape[2]. But displaying the PNG is a little less obvious. We could either re-template our `ItemsControl` so that the map image is part of its visual tree, or we could simply put the map image along with the `ItemsControl` into the same `Grid`. In a real application I'd probably go the more formalized route, but for simplicity, I chose the latter approach for this article:

```
<Grid>
  <Image …/>
  <local:CustomItemsControl …
```

Now our map image will display behind our `ItemsControl`, but our `ItemsControl` will essentially block out the image. We know we're going to need fine-grained control over the coordinates of our items so that we can use the longitude and latitude of the item to position it on the map. We could write our own Mercator projection panel here, and again this is a route I would consider for a real application. But for illustration purposes, I chose to simply use a `Canvas` as the panel for our `ItemsControl`:

```
<ItemsPanelTemplate>
    <Canvas/>
</ItemsPanelTemplate>
```

Now we can position our items absolutely using the attached properties that `Canvas` provides us:

```
<local:CustomItemsControl.
ItemContainerStyle>
 <Style TargetType="ContentControl">
  <Setter Property="Canvas.Left">
   <Setter.Value>
    <MultiBinding
    Converter="{StaticResource
    LongitudeConverter}">
     <Binding ElementName="image"
     Path="ActualWidth"/>
     <Binding Path="Longitude"/>
    </MultiBinding>
```

2 https://inkscape.org/

```
  </Setter.Value>
 </Setter>
 <Setter Property="Canvas.Top">
  <Setter.Value>
   <MultiBinding Converter=
   "{StaticResource
   LatitudeConverter}">
    <Binding ElementName="image"
    Path="ActualWidth"/>
    <Binding ElementName="image"
    Path="ActualHeight"/>
    <Binding Path="Latitude"/>
   </MultiBinding>
  </Setter.Value>
 </Setter>
```

Rather than going into the details of the converters, I'll leave it as an exercise for the reader to investigate how to convert longitudes and latitudes to X and Y locations in a Mercator projection. It's just some simple math.

Lastly, I tweaked the item templates so that countries just display their flag and cities display small red dots. The end result is shown in Figure 12.

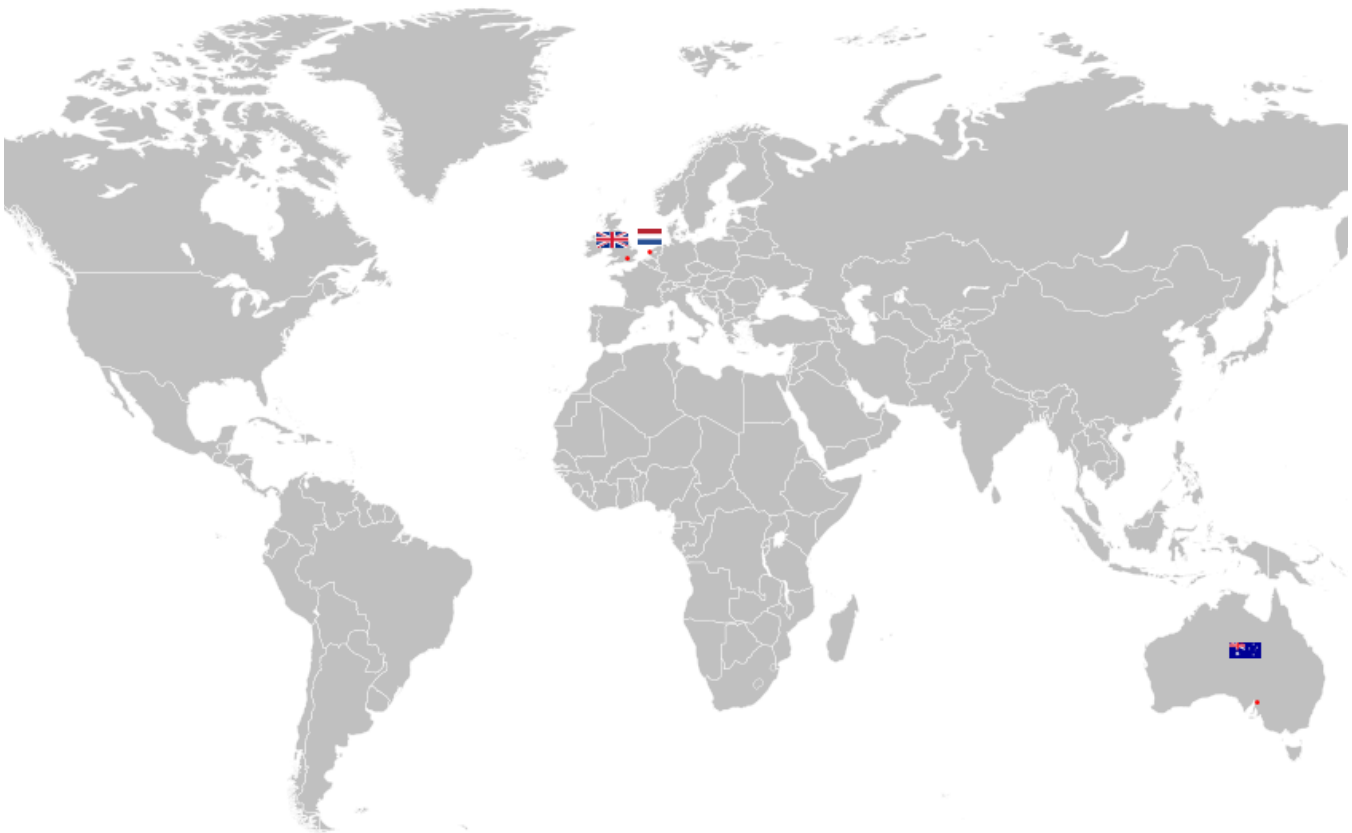


Figure 12: Our items displayed on top of a map

That's a significantly different visualization of the same data, but still utilizing an `ItemsControl`. But what if we want the user to be able to select a place? Looking through the properties of `ItemsControl` you may be disappointed to see nothing to help with selection. But that's where `ItemsControl` subclasses come into play.

ItemsControl Subclasses

Having a firm grasp on all the `ItemsControl` concepts and features we've talked about thus far will be invaluable when you need to utilize other list-like controls. Many WPF controls – `ListBox`, `ComboBox`, `TabControl`, `TreeView`, `Menu`, `StatusBar`, and more – all inherit from `ItemsControl`. So in effect you already know how to populate a `Menu`, modify the appearance of the `StatusBar`, or group items in a `ListBox`.

Of course, the `ItemsControl` subclasses do add functionality and behavior according to the problems they're solving. It's too great a task to

cover every piece of behavior added by every `ItemsControl` subclass, but we're going to close out this article by discussing a few of them.

Selection

One glaring omission from `ItemsControl` is the concept of selection. We have no way of telling an `ItemsControl` that a certain item is selected, or how to render a selected item. We could concoct our own solution to this – perhaps with an attached `IsSelected` property and some triggers in our XAML. But the concept of item selection is so frequently required that it is built into WPF, so we'd be wasting our time.

`ItemsControl` subclasses such as `ListBox` and `ComboBox` support selection via an intermediate abstract class called `Selector`. The `Selector` class gives us a number of selection-related properties to track and modify the currently selected items.

As a first step towards adding selection support to our map of places, we can swap out our custom `ItemsControl` subclass for a `ListBox`. Because of our highly customized UI, we could equally have used a `ComboBox`. When using the standard control templates, the choice between these two controls is clearer. The `ListBox` displays all its items in a list, whereas the `ComboBox` shows only the selected item by default, but can display a list if the user requests it. The `ComboBox` also optionally allows direct text entry (hence the name – it is a combination of both a list, and text entry control).

Nothing much changes in our XAML apart from tweaking the `ItemContainerStyle` to target `ListBoxItem`, which is the container type that `ListBox` wraps each item in. We also have to specify a transparent background for our `ListBox` because otherwise our map image won't show through. Once we've made those small changes, we wind up with Figure 12 again, only this time our items can be selected by clicking on them (which can again be verified via the WPF Visualizer). The trouble is, we've provided a custom template for the `ListBoxItem` instances, and that template does not include any visual changes for selected items.

To rectify this, we can modify our `ListBoxItem` template as follows:

```
<Setter Property="Template">
 <Setter.Value>
  <ControlTemplate
  TargetType="ListBoxItem">
   <Border x:Name="border"
   BorderThickness="1" Padding="3">
    <ContentPresenter/>
   </Border>
   <ControlTemplate.Triggers>
    <Trigger Property="IsSelected"
    Value="True">
     <Setter
      TargetName="border"
      Property="BorderBrush"
      Value="#FF60AFFF"/>
     <Setter
      TargetName="border"
      Property="Background"
      Value="#8060AFFF"/>
    </Trigger>
   </ControlTemplate.Triggers>
  </ControlTemplate>
 </Setter.Value>
</Setter>
```

Naturally, there are many ways we could depict selection, but here I have chosen to display a simple blue border around the selected item. Figure 13 shows a zoomed portion of the end result, where Australia has been selected.

Figure 13: Depicting selection with a blue border

Scrolling

If we add more places to our data and revert to displaying our items in a list instead of on a map, we'll find that the `ItemsControl` does not intrinsically support scrolling. That is, we'll wind up either having to give our `ItemsControl` more space, or items will get cut off and the user will have no way of viewing them.

To illustrate this, take a look at Figure 14. On the left is an `ItemsControl` and on the right, a `ListBox`. Both controls are bound to the same data – a list of countries. Notice how the `ListBox` is scrollable (indeed, I have scrolled about a third of the way through the list) whereas the `ItemsControl` only shows the first 20 or so items with the rest out of view.

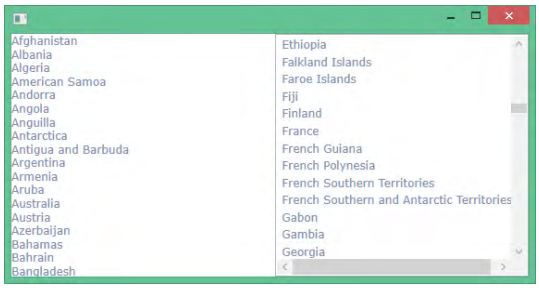


Figure 14: An ItemsControl (left) and ListBox (right) bound to the same data

It's interesting to note that this is not a limitation of `ItemsControl` itself, but rather of the default template for `ItemsControl`. It's entirely possible to re-template `ItemsControl` to include a `ScrollViewer`, thus allowing users to scroll through the items within. If we modify the `ItemsControl` template as follows:

```
<ItemsControl.Template>
  <ControlTemplate TargetType=
  "ItemsControl">
    <ScrollViewer>
      <ItemsPresenter/>
    </ScrollViewer>
  </ControlTemplate>
</ItemsControl.Template>
```

We are now able to scroll through the items within the `ItemsControl`, per Figure 15. However, subclasses of `ItemsControl` typically do this for us because they're specifically designed to be used in such scenarios. In addition to that, they often enable UI virtualization.

Figure 15: Enabling scrolling in ItemsControl via a custom template

UI Virtualization

The idea of UI virtualization, at least in the context of `ItemsControl`, is to pool and re-use item containers. As the user scrolls through items, the containers for those items that are moved out of view can be re-used to host those that are coming into view. This means the total number of containers required to display the view is kept to a minimum, , and less cruft is created for the garbage collector to clean up.

By default, both the `ListBox` and `ListView` controls have UI virtualization enabled. It can be enabled on other controls too – even `ItemsControl` itself, assuming the template for the `ItemsControl` wraps the items in a `ScrollViewer`, as we did above.

The key to enabling UI virtualization on those controls that don't enable it by default is to use a `VirtualizingPanel` for the `ItemsPanel` and set the `IsVirtualizing` property to `true`. There are some things you can do inadvertently that prevent UI virtualization, so beware. For example, adding items manually (as opposed to using data binding) and heterogeneous item containers (e.g. mixing `MenuItem` and `Separator` in the same `Menu`) will both silently disable UI virtualization.

Conclusion

The `ItemsControl` is a real workhorse in WPF and other XAML stacks. Its humble veneer could trick you into overlooking its power and flexibility. This two part article walked you through many aspects

of the `ItemsControl` itself, such as populating data, customizing appearance, and adjusting layout. We then looked at features such as selection and scrolling, which are provided by subclasses of `ItemsControl`.

The detailed understanding of `ItemsControl` you now possess will change the way you think about many common UI problems. Solutions to previously intractable UI scenarios will become more evident to you, especially as you put to practice the techniques presented here ■

Download the entire source code from GitHub at
bit.ly/dncm20-wpf-itemscontrol

About the Author

Kent Boogaart *is a Microsoft MVP in Windows Platform Development since 2009. He lives in Adelaide, Australia with his wife and two kids. You can find his blog at http://kent-boogaart.com and follow him on Twitter @kent_boogaart.*

DIGGING DEEPER INTO HIGH-TRUST APPS IN SHAREPOINT

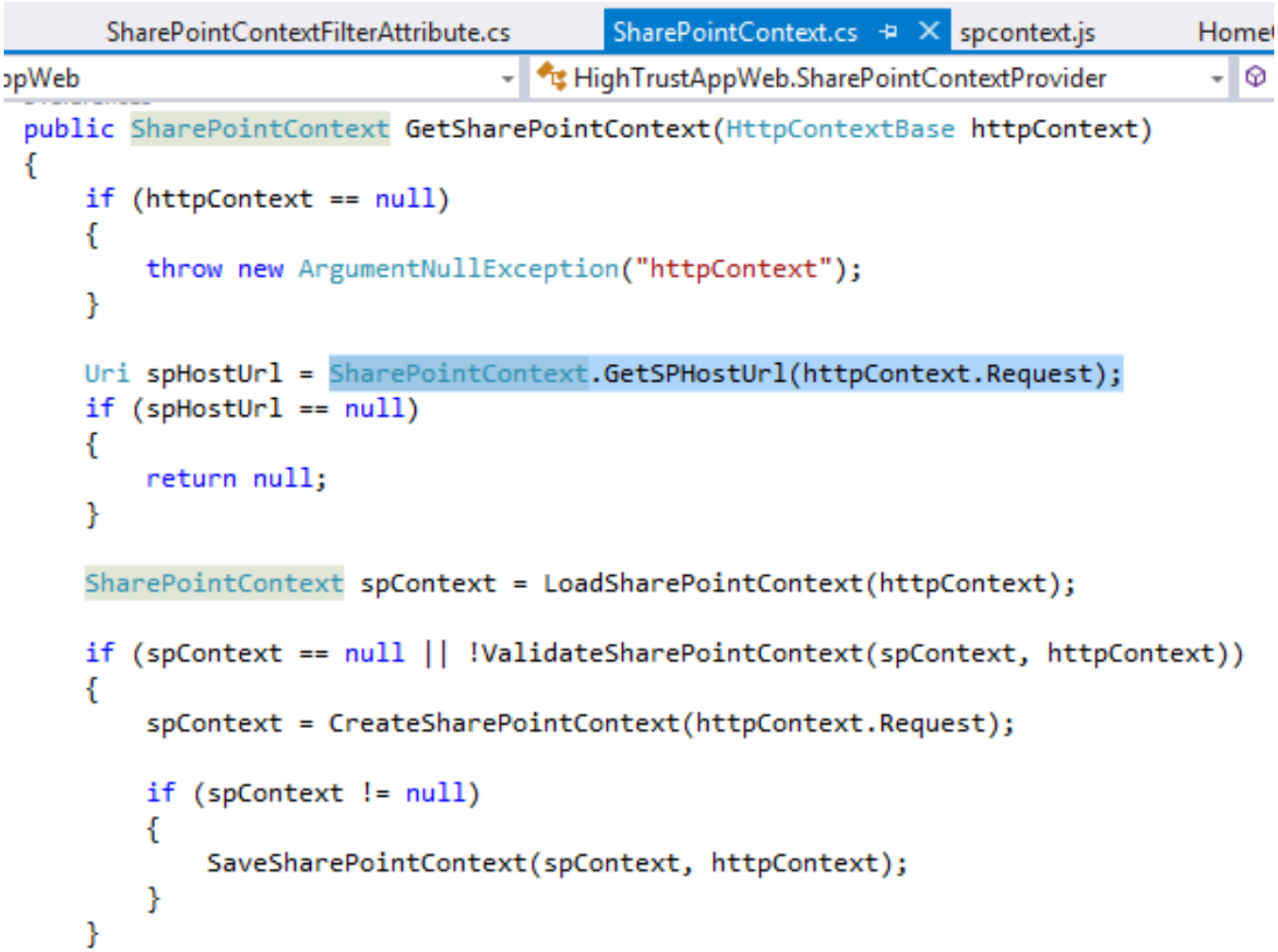
In one of the previous articles, I wrote about high-trust apps in SharePoint 2013. The high-trust apps or server-to-server apps (S2S) are intended to be installed on your SharePoint on-premises datacentre and don't require connectivity to the Internet, unlike low-trust or cloud SharePoint apps such as those available in the SharePoint Store.

In this article I want to go beyond the "Hello, World" example and tackle some real-world issues with high-trust app development.

SharePoint Context Hurdles

The Visual Studio template for SharePoint high-trust app, provisions `SharePointContext.cs` and `TokenHelper.cs` files that hide all the complexity of making the authorized calls to SharePoint. This is achieved using a custom attribute in ASP.NET MVC or `Pre_Init` page event in Web Forms that inspects the current HTTP context, extracts the query string parameters and then makes the app access token with this information. This SharePoint context is stored in the user session, so that it doesn't have to be made again for repeated requests to the app.

The standard app interaction flow implies that the user launches the app from SharePoint, clicking on the app icon. SharePoint then constructs the redirect URL with the app start URL and extra parameters such as the URL of the host web, the URL of the app web (if exists), SharePoint version and navigation bar colour. The browser is then redirected to this long URL and the app uses the extra parameters in the URL to figure out how to make the access token and to which SharePoint endpoint should it make the CSOM or REST call. Even if the SharePoint context instance is stored in the session, the URL parameters have to be present in every HTTP request as the `SharePointContextProvider` classes check in the `CheckRedirectionStatus` method for their presence to validate SharePoint contexts for every incoming request.



Once our app is loaded in the browser, what happens when we click a link in the app that leads to another page or another action? These extra parameters that SharePoint puts in the URL are gone. The standard solution in the Visual Studio template is to use a JS script called `spcontext.js` that transparently appends these parameters in every application link, which is a very clumsy approach in my opinion.

A better way would be to retrieve the parameters in the app landing page, store them somewhere and then reuse them with no need to carry them around. This requires fiddling with the standard `SharePointContext.cs` and `TokenHelper.cs`.

We can even completely dispense with the parameters. How is this possible? In the normal flow, it's SharePoint that provides these parameters. But, it is not the only way.

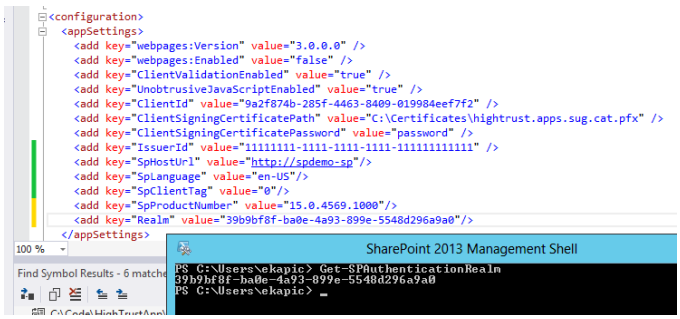
Imagine that instead of going to SharePoint for launching the app, we want to launch the app via its URL. The app would open but without the parameters it would be unable to connect to SharePoint. However, the parameters are nothing magic, just a couple of strings. We can store them in the `web.config` file, for example, and then use them as constants throughout the app code. As high-trust apps run in our own SharePoint farm, we can hardcode the URLs for both the host web (SharePoint site where the app is installed) and the app web (automatically provisioned hidden SharePoint side for the app to store its data). They are known as soon as the app is installed in SharePoint and we can copy them to our app configuration file.

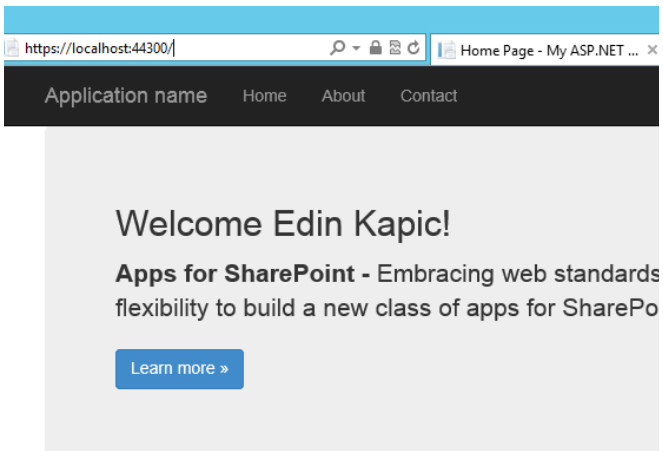
In order to use this technique, we have to change the `SharePointContext.cs` file not to use any query strings but to use `web.config` provided parameters.

Instead of calls to `SharePointContext.GetSPHostUrl` we'll call `WebConfigurationManager.AppSettings["SpHostUrl"]`. Also, in `SharePointContextProvider.CreateSharePointContext` method we'll replace the query string parsing with the calls to `AppSettings` keys.

Another nuisance with the out-of-the-box `TokenHelper.cs` file is that it repeatedly queries SharePoint in the `GetRealmFromTargetUrl` method to obtain the SharePoint realm ID. This ID is a simple Guid and it is needed to make the app access token. In Office 365 world, each SharePoint tenant has a unique realm ID, but in our SharePoint farm, the entire farm is a single realm. Thus, we can also put the realm ID in our app configuration file and skip the unnecessary realm checking altogether.

To get the realm ID, just open a SharePoint PowerShell console and type `Get-SPAuthenticationRealm`.

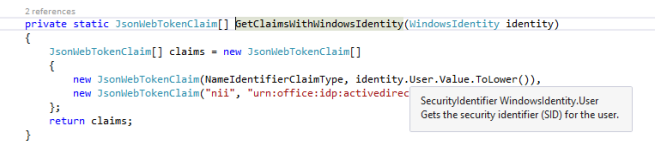




Tokens and Non-Windows Authentication

The high-trust app access token is created by the app itself. The app is responsible for inserting the user ID in the context so that SharePoint can rehydrate the user in the CSOM call and check permissions.

In the default implementation of `TokenHelper` class, it is implied that the app runs with Windows authentication enabled, so the call to `WindowsIdentity` class won't return null. However, with SharePoint, we aren't limited to using only Windows authentication. We can use the broader claims identity instead. As a matter of fact, SharePoint 2013 uses claims authentication by default and `TokenHelper.cs` class converts the Windows identity of the app user into a set of claims that include the current Windows user's security identifier (SID).



While this approach is valid if we only use Windows authentication, when our SharePoint is configured to use alternative identity providers such as ADFS or Azure AD, the default token helper implementation will fail as user SID won't be recognized as a valid user identifier for SharePoint. The exact mechanisms for user mapping between the high-trust app and SharePoint are very well described by Steve Peschka.

Let's say that we want to use ADFS to federate SharePoint with another SAML identity provider. This could be a common scenario for extranet portals, where the external users come from another identity provider such as Microsoft Account, Google or Facebook. We will have to change `SharePointContext.cs` and `TokenHelper.cs` files to take federated identity into account and our app should also include federated identity authentication modules.

The good news is that a fellow SharePoint MVP Wictor Wilén has already done the changes and documented the entire process in his blog.

Caching

The vast majority of the app samples show how to get the data from SharePoint, but very few of them take into account that SharePoint is just another data source for our app, akin to SQL Server. It means that we should call SharePoint sparingly, and keep a local copy of the data that our application needs. In other words, we should use some sort of cache mechanism in our app.

In .NET Framework 4, there is an in-memory object cache implementation, available in MemoryCache class in `System.Runtime.Caching` namespace. It allows for insertion and retrieval of objects by a key, together with cache lifetime and item expiration management. It is very easy to use and even easier to make a caching interface that allows for SharePoint data calls to be wrapped in a lambda expression. By having an interface, we can unit test the code with or without real implementations.

The code for a caching provider and interface is simple. The only detail is that the access to a static variable `MemoryCache.Default` is protected with a lock object to avoid concurrency issues.

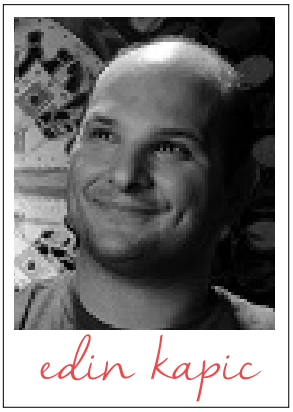
```
public interface ICachingProvider
{
  T GetFromCache<T>(string key, Func<T>
  cacheMissCallback);
}
public class CachingProvider :
ICachingProvider
{
  protected MemoryCache _cache =
  MemoryCache.Default;
  protected static readonly object
  padLock = new object();
  private const int DEFAULT_CACHE_
  LIFETIME_MINUTES = 60;

  private void AddItem(string key,
  object value)
  {
    lock (padLock)
    {
      CacheItem item = new
      CacheItem(key, value);
      CacheItemPolicy policy = new
      CacheItemPolicy();
      policy.AbsoluteExpiration =
      TimeOffset.Now.AddMinutes(DEFAULT_
      CACHE_LIFETIME_MINUTES);
      policy.RemovedCallback = new
      CacheEntryRemovedCallback((args)
      =>
      {
        if (args.CacheItem.Value is
        IDisposable)
        {
          ((IDisposable)args.CacheItem.
          Value).Dispose();
        }
      });
      _cache.Set(item, policy);
    }
  }

  private object GetItem(string key)
  {
    lock (padLock)
    {
      var result = _cache[key];
      return result;
    }
  }

  public T GetFromCache<T>(string key,
  Func<T> cacheMissCallback)
  {
    var objectFromCache = GetItem(key);
    T objectToReturn = default(T);
    if (objectFromCache == null)
    {
      objectToReturn =
      cacheMissCallback();
      if (objectToReturn != null)
      {
        AddItem(key, objectToReturn);
      }
    }
    else
    {
      if (objectFromCache is T)
      {
        objectToReturn = (T)
        objectFromCache;
      }
    }
    return objectToReturn;
  }
}
```

Conclusion

The high-trust apps in SharePoint 2013 are one fine example of using the same model for apps in the cloud and on-premises. However, the implementation details of the code that manages the app communication with SharePoint steps in our way. In real-world scenarios, the default code that ships with Visual Studio project template for SharePoint apps should be extended and optimized. Happy coding! ■

Download the entire source code from GitHub at
bit.ly/dncm20-sharepointhightrust

About the Author



Edin Kapić *is a SharePoint Server MVP since 2013 and works as a SharePoint Architect in Barcelona, Spain.*

He speaks about SharePoint and Office 365 regularly and maintains a blog www.edinkapic.com and Twitter @ekapic.

THANK YOU

FOR THE 20TH EDITION

@gouri_sohoni

@shobankr

@irvindominin

@sravi_kiran

@ekapic

@subodhsohoni

@damirarh

@maheshdotnet

@kent_boogaart

@craigber

@suprotimagarwal

WRITE FOR US

@saffronstroke

OPEN CLOSED PRINCIPLE

Software Gardening: Seeds

Today we visit the letter O, the second principle of SOLID. The O is for the Open-Closed Principle. The Open-Closed Principle (OCP) states "A software Entity should be open for extension but closed to modification". Credit for creating term Open-Closed generally goes to Bertrand Meyer in his 1998 book, "Object Oriented Software Construction".

When you first hear about Open-Closed, you have to wonder, "How can something be open and closed at the same time?" I have heard this jokingly called Shröedinger's OOP Principle. However, Shröedinger was talking about cats, not Object Oriented Programming.

Robert C. "Uncle Bob" Martin further expanded on the definition of OCP in his 2003 book, "Agile Software Development: Principles, Patterns, and Practices":

> "Closed for modification." Extending the behavior of a module does not result in changes to the source or binary code of the module. The binary executable version of the module, whether in a linkable library, a DLL, or a Java .jar, remains untouched.
>
> "Open for extension." This means that the behavior of the module can be extended. As the requirements of the application change, we are able to extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.

Ugh. Sounds like Uncle Bob wants us to do the impossible. Not only can we not change existing code, but we can't change the exe or dll files either. Hang on. I'll explain how this can be done.

Closed for Modification

Let's drill into this principle, first by looking at closed for modification as this one is easier to discuss. In a nutshell, being *closed for modification* means you shouldn't change the behavior of existing code. This ties in nicely with Single Responsibility that I explained in my column in the previous issue. If a class or method does one thing and one thing only, the odds of having to change it are very slim.

There are however, three ways you could change the behavior of existing code.

The first is to fix a bug. After all, the code is not functioning properly and should be fixed. You need to be careful here as clients of the class may know about this bug and have taken steps to get around this. As an example, HP had a bug in some of their printer drivers for many years. This bug caused printing errors under Windows, and HP refused to change it. Microsoft made changes in Word and other applications to get around this bug.

The second reason to modify existing code is to refactor the code so that it follows the other SOLID principles. For example, the code may be working perfectly but does too much, so you wish to refactor it to follow Single Responsibility.

Finally, a third way, which is somewhat controversial, is you are allowed to change the code if it doesn't change the need for clients to change. You have to be careful here so that you don't introduce bugs that affect the client. Good unit testing is critical.

Open for Extension

Having closed to modification out of the way, we turn to *open for extension*. Originally, Meyers discussed this in terms of *implementation inheritance*. Using this solution, you inherit a class and all its behavior then override methods where you want to change. This avoids changing the original code and allows the class to do something different.

Anyone that has tried to do much implementation inheritance knows it has many pitfalls. Because of this many people have adopted the practice *interface inheritance*. Using this methodology, only method signatures are defined and the code for each method must be created each time the interface is implemented. That's the down side. However, the upside is greater and allows you to easily substitute one behavior for another. The common example is a logging class based on an *ILogger* interface. You then implement the class to log to the Windows Event Log or a text file. The client doesn't know which implementation it's using nor does it care.

Another way to have a method open for extension is through *abstract methods*. When inherited, an abstract method must be overridden. In other words, you are required to provide some type of functionality.

Closely related to an abstract method is a *virtual method*. The difference is where you **must** override

the abstract method; you are not required to override a virtual method. This allows a bit more flexibility to you as a developer.

There is one other, and often overlooked, way to provide additional functionality. That is the *extension method*. While it doesn't change the behavior of a method, it does allow you extend the functionality of a class without changing the original class code.

Now, let's return to Uncle Bob's expansion of the definition of closed for modification. We need to extend the functionality of the class without changing the original code or the binary (exe or dll) file. The good news is every method of extending behavior we've looked at here complies with Uncle Bob's definition. Nothing in .NET says that everything about a class has to be in the same source file or even in the same assembly. So, by putting the extended code in a different assembly, you can comply to OCP as defined by Uncle Bob.

An example

Explanations are good, but let's look at an example. First up, code that violates OCP.

```
public class ErrorLogger
{
  private readonly string _whereToLog;
  public ErrorLogger(string whereToLog)
  {
    this._whereToLog = whereToLog.
    ToUpper();
  }

  public void LogError(string message)
  {
    switch (_whereToLog)
    {
      case "TEXTFILE":
        WriteTextFile(message);
        break;
      case "EVENTLOG":
        WriteEventLog(message);
        break;
      default:
        throw new Exception("Unable to
        log error");
    }
  }

  private void WriteTextFile(string
  message)
  {
    System.IO.File.WriteAllText(@"C:\
    Users\Public\LogFolder\Errors.txt",
    message);
  }

  private void WriteEventLog(string
  message)
  {
    string source = "DNC Magazine";
    string log = "Application";

    if (!EventLog.SourceExists(source))
    {
      EventLog.CreateEventSource(source,
      log);
    }
    EventLog.WriteEntry(source, message,
    EventLogEntryType.Error, 1);
  }
}
```

What happens if you need to add a new logging location, say a database or a web service? You need to modify this code in several places. You also need to add new code to write the message to the new location. Finally, you have to modify the unit tests to account for new functionality. All of these types of change have the possibility of introducing bugs.

You may have missed another problem with this code. It violates the Single Responsibility Principle.

Here's the better way. While not fully feature complete, this is a starting point.

```
public interface IErrorLogger
{
  void LogError(string message);
}

public class TextFileErrorLogger :
IErrorLogger
{
  public void LogError(string message)
  {
    System.IO.File.WriteAllText(@"C:\
    Users\Public\LogFolder\Errors.txt",
    message);
  }
}

public class EventLogErrorLogger :
IErrorLogger
{
```

```
  public void LogError(string message)
  {
    string source = "DNC Magazine";
    string log = "Application";

    if (!EventLog.SourceExists(source))
    {
      EventLog.CreateEventSource(source,
      log);
    }

    EventLog.WriteEntry(source, message,
    EventLogEntryType.Error, 1);
  }
}
```

When you need to implement other types of loggers, it's simple…just add new classes rather than modify existing ones.

```
public class DatabaseErrorLogger :
IErrorLogger
{
  public void LogError(string message)
  {
    // Code to write error message to a
    database
  }
}

public class WebServiceErrorLogger :
IErrorLogger
{
  public void LogError(string message)
  {
    // Code to write error message to a
    web service
  }
}
```

Ahhh…a much better way to architect a logging feature.

You now have another seed for your software garden. By following the Open-Closed Principle your code will be better and less prone to bugs. You're on way to making your software lush, green, and vibrant.

About Software Gardening

Comparing software development to constructing a building says that software is solid and difficult to change. Instead, we should compare software development to gardening as a garden changes all the time. Software Gardening embraces practices and tools that help you create the best possible garden for your software, allowing it to grow and change with less effort.

About the Author

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

Building a Sound Cloud Music Player in Windows 10

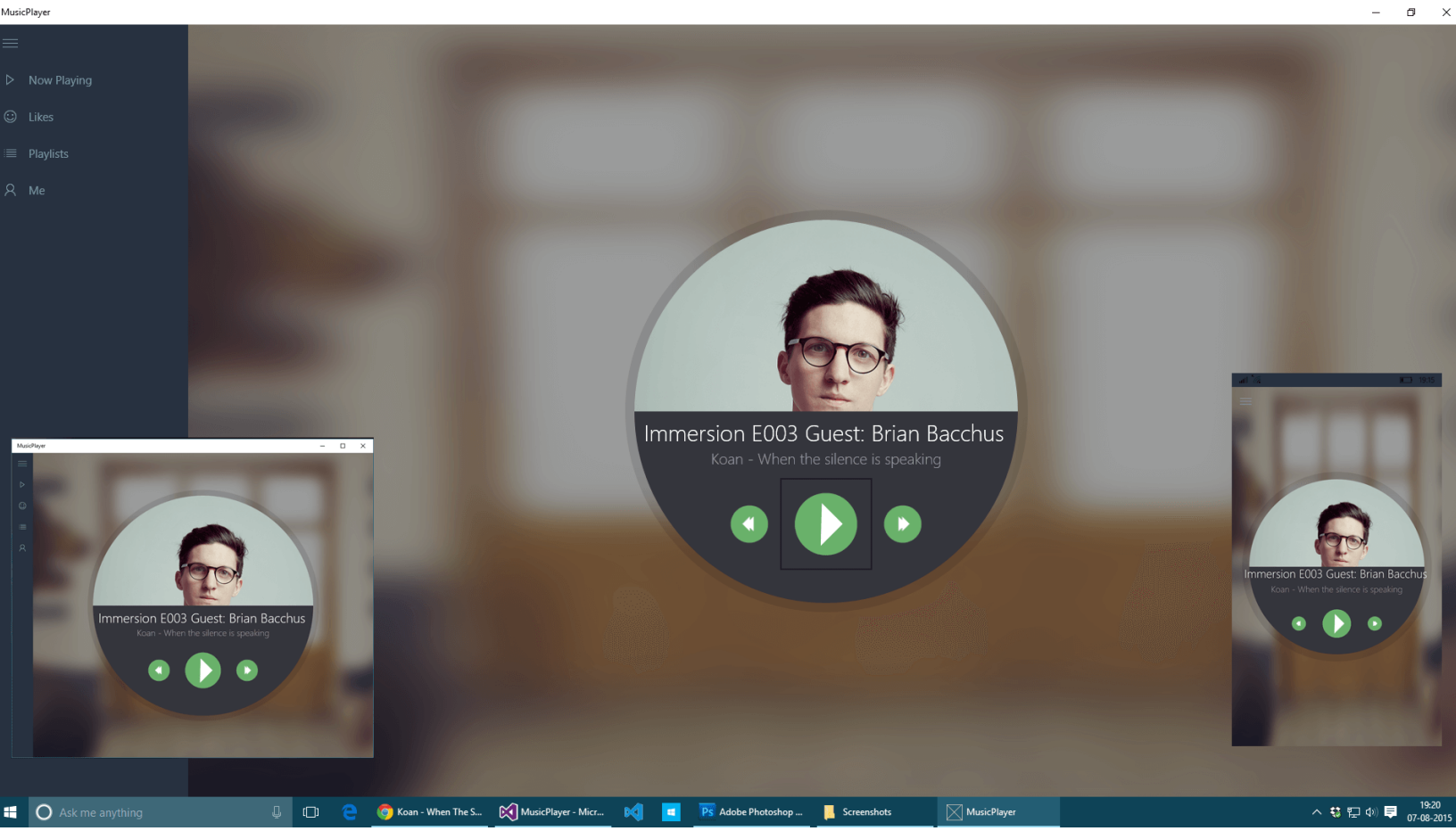
Microsoft's vision of apps that can be written once and will run on a wide variety of devices, is finally a reality with the release of Windows 10. The single, unified Windows core enables one app to run on every Windows device – be it a phone, a tablet, a laptop, a PC, or the Xbox console. And very soon the same apps will also run on the upcoming devices being added to the Windows family, including IoT devices like the Raspberry Pi 2, Microsoft HoloLens and the Surface Hub.

In my previous column "Build a Windows 10 Universal App – First Look" I wrote an introduction about why Windows 10 is different from previous versions of Windows and how you, as a developer, can take advantages of new features in Windows 10 and write better code.

In this article, we will start developing a better Music Player app and we will also try to use some of the unique new platform features in Windows 10 to develop a single app which runs on multiple devices.

First Look

We are going to develop a Music Player in Windows 10 which will use Sound Cloud API to play our Likes, Playlists etc. Here is a screenshot of how the player will look in Desktop, Tablet and Phone.



If you do not use Sound Cloud (there is no reason not to use Sound Cloud if you love Music), register for a new account and start *Liking* tracks and create playlists for your favourite tracks. We will be using these for our App.

Register a new Client Application by visiting the Sound Cloud Developer Website (https://developers.soundcloud.com/). Keep a note of the Client Id Value.

Project

Fire up Visual Studio 2015 and create a New Blank App (Universal Windows) under *Templates* > *Visual C#* > *Windows*. Right click the new project and click on Manage NuGet Packages. Search for *Newtonsoft.json* library and install the package. You may also want to add some images that are included with the Source Code of this article. We will be using those to design the App.

If you want to learn more about new Controls in Windows 10, I would recommend downloading Developer Samples from GitHub (https://github.com/Microsoft/Windows-universal-samples) and playing around with the sample projects.

Shell

One of the new controls introduced in Windows 10 is the SplitView. It is a very simple but useful control which can be used to build quick Navigation related features in our App. This control is also very flexible in terms of customization and presents unique experiences across different devices.

We will change the new project to use a Custom Page that will hold the SplitView control and this page will act as the shell for the whole app giving it a Single Page App like experience.

- Add a Blank Page by Right Clicking the Project > Add > New Item. Name this new Page as *AppShell.xaml*

- Add the following xaml code to AppShell.xaml

```
<Page.Resources>
 <DataTemplate
 x:Key="NavMenuItemTemplate"
 x:DataType="local:NavMenuItem" >
  <Grid>
   <Grid.ColumnDefinitions>
    <ColumnDefinition Width="48" />
    <ColumnDefinition />
   </Grid.ColumnDefinitions>
```

```
    <FontIcon x:Name="Glyph"
    FontSize="16" Glyph="{x:Bind
    SymbolAsChar}"
    VerticalAlignment="Center"
    HorizontalAlignment="Center"
    ToolTipService.ToolTip="{x:Bind
    Label}"/>

    <TextBlock x:Name="Text" Grid.
    Column="1" Text="{x:Bind Label}" />
   </Grid>
  </DataTemplate>
</Page.Resources>

<Grid Background="#273140">
 <!-- Adaptive triggers -->
 <VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
   <VisualState>
    <VisualState.StateTriggers>
     <AdaptiveTrigger
     MinWindowWidth="720" />
    </VisualState.StateTriggers>
    <VisualState.Setters>
     <Setter Target="RootSplitView.
     DisplayMode" Value="CompactInline"/>
     <Setter Target="RootSplitView.
     IsPaneOpen" Value="True"/>
    </VisualState.Setters>
   </VisualState>
   <VisualState>
    <VisualState.StateTriggers>
     <AdaptiveTrigger MinWindowWidth="0"/>
    </VisualState.StateTriggers>
    <VisualState.Setters>
     <Setter Target="RootSplitView.
     DisplayMode" Value="Overlay"/>
    </VisualState.Setters>
   </VisualState>
  </VisualStateGroup>
 </VisualStateManager.VisualStateGroups>

 <!-- Top-level navigation menu + app
 content -->
 <SplitView x:Name="RootSplitView"
  DisplayMode="Inline"
  OpenPaneLength="256"
  IsTabStop="False"
  Background="#273140"
 >
 <SplitView.Pane>
  <Grid Background="#273140">
   <!-- A custom ListView to display the
   items in the pane. -->

   <controls:NavMenuListView
   x:Name="NavMenuList"
   Background="#273140"
   TabIndex="3"
   Margin="0,48,0,0"
   ItemContainerStyle="{StaticResource
   NavMenuItemContainerStyle}"
   ItemTemplate="{StaticResource
   NavMenuItemTemplate}"
   ItemInvoked=
   "NavMenuList_ItemInvoked" >
  </controls:NavMenuListView>
 </Grid>

</SplitView.Pane>

<Frame x:Name="frame"
x:FieldModifier="Public">
 <Frame.ContentTransitions>
  <TransitionCollection>
   <NavigationThemeTransition>
    <NavigationThemeTransition.
     DefaultNavigationTransitionInfo>
     <EntranceNavigationTransitionInfo/>
    </NavigationThemeTransition.
    DefaultNavigationTransitionInfo>
   </NavigationThemeTransition>
  </TransitionCollection>
 </Frame.ContentTransitions>
</Frame>
</SplitView>

<ToggleButton x:Name="TogglePaneButton"
Style="{StaticResource
SplitViewTogglePaneButtonStyle}"

IsChecked="{Binding IsPaneOpen,
ElementName=RootSplitView, Mode=TwoWay}"

AutomationProperties.Name="Menu"
ToolTipService.ToolTip="Menu"/>

<MediaElement x:FieldModifier="Public"
AudioCategory="BackgroundCapableMedia"
x:Name="mPlayer"
RequestedTheme="Default"
CompositeMode="MinBlend"/>
</Grid>
```

In the above code, we have added the following:

1. SplitView control (*RootSplitView*) which has a custom Navigation control (*NavMenuList*) in its Pane and a Frame (*frame*) which will hold our Pages.

2. ToggleButton (*TogglePaneButton*) which will be the Hamburger menu button.

3. Data template for binding items for the custom Navigation Control.

4. Common MediaElement control (*mPlayer*) that will be used to play the music.

We also use some custom style templates which we will add shortly to our project.

- Add a new folder named *Controls* and add a class file in it and name it *NavMenuItem.cs* . Add the following code to the newly created file.

```
/// <summary>
 /// Data to represent an item in the
 nav menu.
 /// </summary>
 public class NavMenuItem
 {
  public string Label { get; set; }
  public Symbol Symbol { get; set; }
  public char SymbolAsChar
  {
   get
   {
    return (char)this.Symbol;
   }
  }

  public Type DestPage { get; set; }
  public object Arguments { get; set; }
 }
}
```

The above class represents a Menu Item which holds the Item title, Icon and destination page.

- Add the following code to AppShell.xaml.cs

```
// Declare the top level nav items
 private List<NavMenuItem> navlist = new
 List<NavMenuItem>(
  new[]
  {
   new NavMenuItem()
   {
    Symbol = Symbol.Play,
    Label ="Now Playing",
    DestPage = typeof(NowPlaying)
   },
   new NavMenuItem()
   {
    Symbol = Symbol.Emoji2,
    Label = "Likes",
    DestPage = typeof(Likes)
   },
   new NavMenuItem()
   {
    Symbol = Symbol.List,
    Label = "Playlists",
    DestPage = typeof(MainPage)
   },
   new NavMenuItem()
   {
    Symbol = Symbol.Contact,
    Label = "Me",
    DestPage = typeof(MainPage)
   }
  });

public static AppShell Current = null;

/// <summary>
/// Initializes a new instance of
the  AppShell, sets the static
'Current' reference,
/// adds callbacks for Back requests
and changes in the SplitView's
DisplayMode, and
/// provide the nav menu list with the
data to display.
/// </summary>
public AppShell()
{
 this.InitializeComponent();

 this.Loaded += (sender, args) =>
 {
  Current = this;
 };
 NavMenuList.ItemsSource = navlist;
}

public Frame AppFrame { get { return
this.frame; } }

/// <summary>
/// Navigate to the Page for the
selected <paramref name=
"listViewItem"/>.
/// </summary>
/// <param name="sender"></param>
/// <param name="listViewItem">
</param>

private void NavMenuList_ItemInvoked
(object sender, ListViewItem
listViewItem)
{
 var item = (NavMenuItem)
 ((NavMenuListView)sender).
 ItemFromContainer(listViewItem);

 if (item != null)
 {
  if (item.DestPage != null &&
  item.DestPage != this.AppFrame.
  CurrentSourcePageType)
```

```
    {
     this.AppFrame.Navigate(item.
     DestPage, item.Arguments);
    }
   }
  }

  private void Page_Loaded(object
  sender, RoutedEventArgs e)
  {
   ((Page)sender).Focus(FocusState.
   Programmatic);
   ((Page)sender).Loaded -= Page_Loaded;
  }
 }
```

In the above code, we do the following

1. We create a new List of objects of *NavMenuItem* class and populate it with required menu options and suitable icons.

2. We also add an Item Invoked event handler to handle click events and navigate to subpages.

- If you notice, we are using some custom styles for our controls and we will use a common styles page to hold these style templates. Add a new folder to the Project named *Styles* and add a new *Resource Dictionary* file and name it Styles.xaml. Add the following code to the newly created file (code truncated for brevity. Check the source code)

```
<Style
x:Key="SplitViewTogglePaneButtonStyle"
TargetType="ToggleButton">
 <Setter Property="FontSize" Value="20"
 />
 <Setter Property="FontFamily"
 Value="{ThemeResource
 SymbolThemeFontFamily}" />
 ...
 <Setter Property=
 "UseSystemFocusVisuals" Value="True"/>
 <Setter Property="Template">
  <Setter.Value>
   <ControlTemplate
   TargetType="ToggleButton">
    <Grid Background="{TemplateBinding
    Background}" x:Name="LayoutRoot">
     <VisualStateManager.
     VisualStateGroups>
     <VisualStateGroup x:Name=
     "CommonStates">
      <VisualState x:Name="Normal" />
      <VisualState x:Name="PointerOver">
       <Storyboard>
       ...
       </Storyboard>
      </VisualState>
      <VisualState x:Name="Pressed">
       <Storyboard>
       ...
       </Storyboard>
      </VisualState>
      <VisualState x:Name="Disabled">
       <Storyboard>
       ...
       </Storyboard>
      </VisualState>
      <VisualState x:Name="Checked"/>
      <VisualState x:Name=
      "CheckedPointerOver">
       <Storyboard>
       ...
       </Storyboard>
      </VisualState>
      <VisualState x:Name=
      "CheckedPressed">
       <Storyboard>
       ...
       </Storyboard>
      </VisualState>
      <VisualState x:Name=
      "CheckedDisabled">
       <Storyboard>
       ...
       </Storyboard>
      </VisualState>
     </VisualStateGroup>
    </VisualStateManager.
    VisualStateGroups>

    <ContentPresenter x:Name=
    "ContentPresenter"
    Content="{TemplateBinding Content}"
    Margin="{TemplateBinding Padding}"
    HorizontalAlignment=
    "{TemplateBinding
    HorizontalContentAlignment}"

     VerticalAlignment
     ="{TemplateBinding
     VerticalContentAlignment}"
     AutomationProperties.
     AccessibilityView="Raw" />
    </Grid>
   </ControlTemplate>
  </Setter.Value>
 </Setter>
</Style>
<Style x:Key="PageTitleTextBlockStyle"
TargetType="TextBlock"
BasedOn="{StaticResource
```

```
BodyTextBlockStyle}">
...
</Style>

<Style x:Key="NavMenuItemContainerStyle"
TargetType="ListViewItem">
 <Setter Property="MinWidth"
 Value="{StaticResource
 SplitViewCompactPaneThemeLength}"/>
 <Setter Property="Height" Value="48"/>
 <Setter Property="Padding" Value="0"/>
 <Setter Property="Background"
 Value="#273140"/>
 <Setter Property="Foreground"
 Value="#7f96a3"/>
 <Setter Property="Template">
  <Setter.Value>
  ...
  </Setter.Value>
 </Setter>
</Style>
```

- Change the OnLaunched event handler code in App.xaml.cs to the following to use our custom AppShell and not the default Frame

```
protected override void
OnLaunched(LaunchActivatedEventArgs e)
{
 #if DEBUG
 if (System.Diagnostics.Debugger.
 IsAttached)
 {
  this.DebugSettings.
  EnableFrameRateCounter = true;
 }
 #endif
  AppShell shell = Window.Current.
  Content as AppShell;

  // Do not repeat app initialization
  when the Window already has content,
  // just ensure that the window is
  active
  if (shell == null)
  {
   // Create a Frame to act as the
   navigation context and navigate to
   the first page
   shell = new AppShell();

   // Set the default language
   shell.Language = Windows.
   Globalization.ApplicationLanguages.
   Languages[0];
   shell.AppFrame.NavigationFailed +=
   OnNavigationFailed;
   if (e.PreviousExecutionState ==
   ApplicationExecutionState.Terminated)
   {
    //TODO: Load state from previously
    suspended application
   }
  }

  // Place the frame in the current
  Window
  Window.Current.Content = shell;

  if (shell.AppFrame.Content == null)
  {
   // When the navigation stack isn't
   restored, navigate to the first page
   // suppressing the initial entrance
   animation.
   shell.AppFrame.
   Navigate(typeof(MainPage),
   e.Arguments, new Windows.
   UI.Xaml.Media.Animation.
   SuppressNavigationTransitionInfo());
  }
 // Ensure the current window is active
 Window.Current.Activate();
}
```

- Add the following code to App.xaml.cs. These variables will hold values that will be used in other pages.

```
public static string SoundCloudClientId
= "<INSERT YOUR CLIENT ID";
public static int SCUserID = 0;
public static string SoundCloudLink =
"http://api.soundcloud.com/";

public static string SoundCloudAPIUsers
= "users/";
public static List<SoundCloudTrack>
nowPlaying = new
List<SoundCloudTrack>();
public static int nowplayingTrackId = 0;
```

Sound Cloud API

If you are new to Sound Cloud API, I would recommend going through the HTTP API Reference on this page (https://developers.soundcloud.com/docs/api/reference). SoundCloud objects (sounds, users etc) can be accessed by using HTTP methods GET, POST, PUT and DELETE and the response is a JSON string.

We will be using Json.NET which is a high performance, popular JSON framework to serialize and deserialize Sound Cloud objects from Sound Cloud website and custom Sound Cloud objects in our App. I used the website http://json2csharp.com/ to quickly create classes (and modified for our app) for Sound Cloud objects that we will be adding in our next step.

- Add the following new class files to our project

o SoundCloudTrack.cs

```
public class SoundCloudTrack{
 public int id { get; set; }
 public string created_at { get; set; }
 public int user_id { get; set; }
 public int duration { get; set; }
 ...
 public SoundCloudUser user { get; set;
}
 ...
 public SoundCloudCreatedWith created_
 with { get; set; }
 public string attachments_uri { get;
 set; }
}
```

o SoundCloudCreatedWith.cs

```
public class SoundCloudCreatedWith
{
 public int id { get; set; }
 public string name { get; set; }
 public string uri { get; set; }
 public string permalink_url { get;
 set;}
}
```

o SoundCloudUser.cs

```
public class SoundCloudUser
{
 public int id { get; set; }
 public string permalink { get; set; }
 public string username { get; set; }
 public string uri { get; set; }
 public string permalink_url { get;
 set; }
 public string avatar_url { get; set; }
}
```

Landing Page

We will use *MainPage.xaml* as the landing page for our app which will handle login and navigate to subsequent pages.

- Add the following code to *MainPage.xaml* .

```
<Grid Background="#edf3fb">
 <StackPanel VerticalAlignment="Center">
  <ProgressRing x:Name="loginProgress"
  HorizontalAlignment="Center"
  IsActive="True" Foreground="#273140"
/>

<TextBlock HorizontalAlignment="Center"
 Text="Please wait..."
 Style="{StaticResource
 SubtitleTextBlockStyle}"
 Foreground="#273140"/>
 </StackPanel>
</Grid>
```

In the above code we add a *ProgressRing* control and *TextBlock* to show the progress.

- Add the following code to *MainPage.xaml.cs*

```
using Newtonsoft.Json;
...

// The Blank Page item template is
documented at http://go.microsoft.com/
fwlink/?LinkId=402352&clcid=0x409

namespace MusicPlayer
{
 /// <summary>
 /// An empty page that can be used on
 its own or navigated to within a Frame.
 /// </summary>
 public sealed partial class MainPage :
 Page
 {
  public MainPage()
  {
   this.InitializeComponent();

   this.Loaded += MainPage_Loaded;
  }

  private void MainPage_Loaded(object
  sender, RoutedEventArgs e)
  {
```

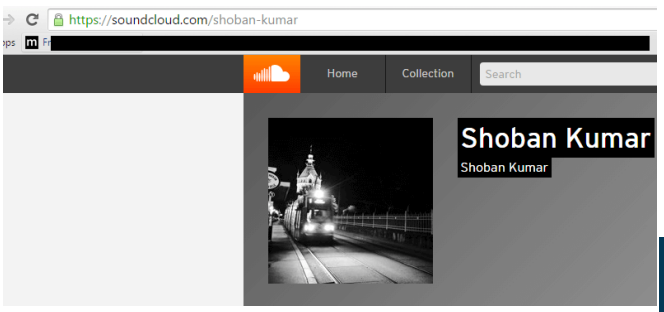
```
 //Get User
 GetUserDetails();
}

private async void GetUserDetails()
{
 try
 {
  string responseText = await
  GetjsonStream(App.SoundCloudLink
  + App.SoundCloudAPIUsers +
  "shoban-kumar" + ".json?client_id="
  + App.SoundCloudClientId);
  SoundCloudUser user = JsonConvert.
  DeserializeObject<SoundCloudUser>
  (responseText);

  App.SCUserID = user.id;

  //Get Likes
  GetLikes();
 }
 catch (Exception ex)
 {
  MessageDialog showMessgae = new
  MessageDialog("Something went
  wrong. Please try again.
  Error Details : " + ex.Message);
  await showMessgae.ShowAsync();
 }
}

private async void GetLikes()
{
 try
 {
  string responseText = await
  GetjsonStream(App.SoundCloudLink +
  App.SoundCloudAPIUsers
  + App.SCUserID + "/favorites.
  json?client_id=" + App.
  SoundCloudClientId);
  List<SoundCloudTrack> likes
  = JsonConvert.DeserializeObject<List
  <SoundCloudTrack>>(responseText);
  App.nowPlaying = likes;

  loginProgress.IsActive = false;

  AppShell shell = Window.Current.
  Content as AppShell;
  shell.AppFrame.
  Navigate(typeof(NowPlaying));
 }
 catch (Exception ex)
 {
  MessageDialog showMessgae = new
  MessageDialog("Something went wrong.
  Please try again. Error Details : "
   + ex.Message);
   await showMessgae.ShowAsync();
  }

 }

 public async Task<string>
 GetjsonStream(string url) //Function
 to read from given url
 {
  HttpClient client = new HttpClient();
  HttpResponseMessage response = await
  client.GetAsync(url);
  HttpResponseMessage v = new
  HttpResponseMessage();
  return await response.Content.
  ReadAsStringAsync();
 }
}
}
```

In the above code, we are doing the following:

o Fetch user details by sending a request to SoundCloud API and fetch user Id which will be used by future requests to Sound Cloud. The user name (*shoban-kumar*) is used in the sample code

```
string responseText = await
GetjsonStream(App.SoundCloudLink +
App.SoundCloudAPIUsers + "shoban-
kumar" + ".json?client_id=" + App.
SoundCloudClientId);
```

This username is unique and you can get your user name from navigating to your profile page and checking the url for the value.



- Once the user id is retrieved, we GET the list of tracks that are liked by you using *GetLikes* method.

- Once the list of tracks are retrieved, we populate the common variable *nowPlaying* which is a list of SoundCloudTrack objects.

- We also navigate to a new page called NowPlaying which we will add in our next step.

Now Playing

Add a new Blank Page to our project and name it *NowPlaying.xaml*. Add the following xaml code to the page.

```
<Grid Background="#edf3fb">
 <!-- Adaptive triggers -->
 <VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
   <VisualState>
    <VisualState.StateTriggers>
     <AdaptiveTrigger
     MinWindowWidth="720" />
    </VisualState.StateTriggers>
    <VisualState.Setters>
     ...
    </VisualState.Setters>
   </VisualState>
   <VisualState>
    <VisualState.StateTriggers>
     <AdaptiveTrigger MinWindowWidth="0"
     />
    </VisualState.StateTriggers>
    <VisualState.Setters>
     ...
   </VisualState.Setters>
  </VisualState>
 </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
<Image Source="Assets/BG.png"
Stretch="Fill"/>

<Grid VerticalAlignment="Center">
 <Grid.RowDefinitions>
  <RowDefinition Height="Auto"/>
 </Grid.RowDefinitions>
 <!-- Black ring around Album art with
 opacity .10 -->
 <Ellipse Height="525" Grid.Row="0"
 Width="525" Fill="Black"
 Opacity=".10" x:Name="AlbumartRing"/>

 <!-- Album Art -->
  <Ellipse Height="500" Grid.Row="0"
  Width="500" x:Name="Albumart" >
   <Ellipse.Fill>
    <ImageBrush x:Name="albumrtImage"
    ImageSource="Assets\Albumart.png"
    Stretch="UniformToFill" />
   </Ellipse.Fill>
   <Ellipse.Clip >
    <RectangleGeometry
     Rect="0,0,500,250"
     x:Name="AlbumartClip"/>
   </Ellipse.Clip>
  </Ellipse>

  <!-- Bottom part of album art with
  controls -->
  <Ellipse Height="500"
  Grid.Row="0" Width="500"
  x:Name="AlbumartContainerDown"
  Fill="#34353c" >
  <Ellipse.Clip >
   <RectangleGeometry
   x:Name="AlbumartContainerDownClip"
   Rect="0,250,500,500" />
  </Ellipse.Clip>
 </Ellipse>

 <TextBlock x:Name="txtSongTitle" Grid.
 Row="0" HorizontalAlignment="Center"
 Text="Song Title "
 FontSize="25" Foreground="White"
 Style="{StaticResource
 HeaderTextBlockStyle}"
 TextTrimming="WordEllipsis"  />

 <TextBlock x:Name="txtAlbumTitle"
 Grid.Row="0" Text="Label "
 HorizontalAlignment="Center"
 FontWeight="Light" FontSize="20"
 Foreground="#9799a5"
 Style="{StaticResource
 BodyTextBlockStyle}"
 TextTrimming="WordEllipsis"/>

 <StackPanel Margin="0,350,0,0"
 x:Name="ControlContainer" Grid.
 Row="0" Orientation="Horizontal"
 VerticalAlignment="Top"
 HorizontalAlignment="Center">

  <Button x:Name="btnPrev"
   Background="Transparent" Height="80"
   Width="80" Click="btnPrev_Click">
   <Image  Source="Assets/Prev.png"
   Stretch="Uniform"/>
  </Button>

  <Button x:Name="btnPlay"
   Background="Transparent" Height="120"
   Width="120" Click="btnPlay_Click">
   <Image Source="Assets/Play.png"
   Stretch="Uniform"/>
  </Button>

  <Button x:Name="btnNext"
   Background="Transparent" Height="80"
   Width="80" Click="btnNext_Click">
   <Image  Source="Assets/Next.png"
```

```
   Stretch="Uniform"/>
  </Button>
 </StackPanel>

 </Grid>
</Grid>
```

In the code, we add the following controls to build our player user Interface.

- Three Ellipse controls
- ImageBrush control which will hold a default album art picture
- Two TextBlocks to display Song Title and User name
- Three buttons which will act as Play, Next and Previous (We will be adding more buttons later)
- and two VisualStateTriggers for different widths.

These triggers change few properties of controls when the width of the App is changed and this helps us give different experience for different devices without writing any additional C# code.

Add the following code to NowPlaying.xaml.cs

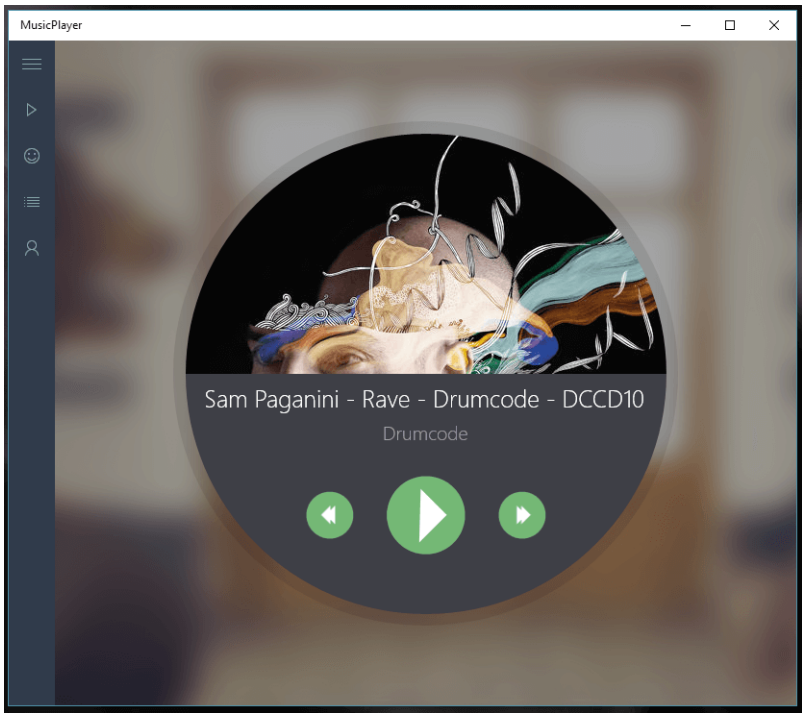
```
namespace MusicPlayer
{
 /// <summary>
 /// An empty page that can be used on
 its own or navigated to within a Frame.
 /// </summary>
 public sealed partial class NowPlaying
 : Page
 {
  AppShell shell = Window.Current.
  Content as AppShell;
  public NowPlaying()
  {
   this.InitializeComponent();
   this.Loaded += NowPlaying_Loaded;
  }

  private void NowPlaying_Loaded(object
  sender, RoutedEventArgs e)
  {
   if (App.nowPlaying.Count > 0)
   {
    SoundCloudTrack currentTrack = App.
    nowPlaying[App.nowplayingTrackId];
    LoadTrack(currentTrack);
   }
  }

  private async void
  LoadTrack(SoundCloudTrack currentTrack)
  {
   try
   {
    //Stop player, set new stream uri and
    play track
    shell.mPlayer.Stop();
    Uri streamUri = new Uri(currentTrack.
    stream_url + "?client_id=" + App.
    SoundCloudClientId);
    shell.mPlayer.Source = streamUri;
    shell.mPlayer.Play();

    //Change album art
    string albumartImage = Convert.
    ToString(currentTrack.artwork_url);
    if (string.
    IsNullOrWhiteSpace(albumartImage))
    {
     albumartImage = @"ms-appx:///Assets\
     Albumart.png";
    }
    else
    {
     albumartImage = albumartImage.
     Replace("-large", "-t500x500");
    }

    albumrtImage.ImageSource = new
    BitmapImage(new Uri(albumartImage));

    //Change Title and User name
    txtSongTitle.Text = currentTrack.
    title;
    txtAlbumTitle.Text = Convert.
    ToString(currentTrack.user.username);

   }
   catch (Exception ex)
   {
    MessageDialog showMessgae = new
    MessageDialog("Something went wrong.
    Please try again. Error Details : " +
    ex.Message);
    await showMessgae.ShowAsync();
   }
  }

  private void btnPlay_Click(object
  sender, RoutedEventArgs e)
  {
  }
```

```
    private void btnNext_Click(object
    sender, RoutedEventArgs e)
    {
     App.nowplayingTrackId += 1;
     if (App.nowplayingTrackId >= App.
     nowPlaying.Count)
     {
      App.nowplayingTrackId = 0;
     }

     SoundCloudTrack currentTrack = App.
     nowPlaying[App.nowplayingTrackId];
     LoadTrack(currentTrack);

     }

     private void btnPrev_Click(object
     sender, RoutedEventArgs e)
     {
      App.nowplayingTrackId -= 1;
      if (App.nowplayingTrackId < 0)
      {
       App.nowplayingTrackId = App.
       nowPlaying.Count - 1;
      }

      SoundCloudTrack currentTrack = App.
      nowPlaying[App.nowplayingTrackId];
      LoadTrack(currentTrack);
     }
    }
}
```

In the above code, once the page is loaded, we start playing the first Track in the *nowPlaying* list and set the Title, User name and album art based on the Track properties that we fetched from Sound Cloud.

Press F5 to debug the app and if there are no compile errors, you should login without errors and our app should start playing the first track from your Likes list in Sound Cloud.

Play around with the width of the app or deploy the app on different devices like Tablets and Phone to see how Adaptive Triggers change control appearances based on how you use the app.
You will notice that minimizing our app will stop the music and pressing volume controls will not show the default media controls. We will change this behaviour and add more features in future articles.



Conclusion

With the introduction of new controls and features like Adaptive Triggers, the Universal Windows app experience has improved significantly for Windows 10 in comparison to the Windows 8.1 experience. You now have to build just one Universal Windows app that will run on all Windows 10 devices ■

Download the entire source code from GitHub at
bit.ly/dncm20-soundcloudwin10

About the Authors

shoban kumar

Shoban Kumar *is an ex-Microsoft MVP in SharePoint who currently works as a SharePoint Consultant. You can read more about his projects at http://shobankumar.com. You can also follow him in Twitter @shobankr*

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

- ASP.NET
- SHAREPOINT
- JAVASCRIPT
- PATTERNS
- AZURE
- VISUAL STUDIO
- .NET
- C#, WPF

We've got it all!

65K PLUS READERS

180 PLUS AWESOME ARTICLES

19 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

NEW BUILD FEATURES IN TFS 2015 AND VISUAL STUDIO ONLINE

Overview

Team Foundation Server had introduced Build Automation right from its very first release i.e. TFS 2005. At that time, the build process was based upon MSBuild technology. There was a major change in Build Service with the introduction of Visual Studio 2010 and Team Foundation Server 2010. Microsoft also introduced a very useful trigger called as Gated Check-in. The newly introduced build process was completely XAML based. This approach was followed until Visual Studio 2013. The XAML build definition could be created with the help of a wizard and it was a straight forward process if you didn't need any customization. Any customization could be done by customizing the default workflow, editing it, as required. We could create a new activity either by using XAML or by writing code and implementing it in a customized process template.

All this required the knowledge of XAML workflow as well as handling the problems and limitations of the XAML builds, one of the issues being builds being able to run on-your-machine but not on the Continuous Integration server. Team Foundation Server 2015, brings a major improvement in the build service. The new build runs on a different architecture and also runs on a completely different system. You can still use your XAML builds along with your new builds, controllers and agents.

In this article, we will discuss the new build features available with Visual Studio 2015/TFS 2015 and Visual Studio Online.

Working with TFS 2015 Build

Major changes

- You do not need to create build, you can add steps for MSBuild, Ant, Maven, PowerShell etc. (readymade templates are also provided)

- You can build cross platform solutions

- You can create custom activity with open source activities

- Live console view is available

We can use either Visual Studio Online account which will give us hosted build controller or on-premises Team Foundation Server 2015 with which we can configure default build controller.
With Visual Studio 2015 you can build for any platform (Windows, iOS, Android, Java or Linux). Just open your browser and start creating and triggering builds.

Visual Studio 2015 shows two different build definitions you can work with - New as well as XAML build definition. The screenshots attached are with Visual Studio 2015 as well as from Visual Studio Online (termed as VSO)

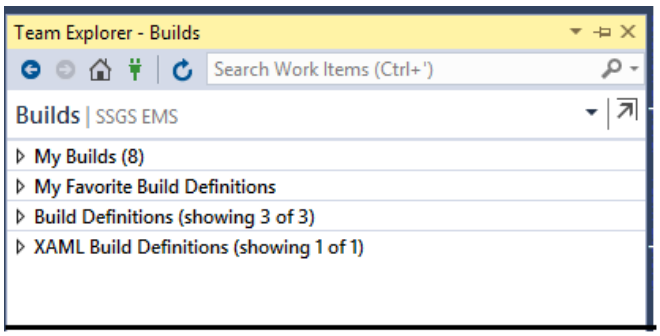


Figure 1: Build Explorer from Visual Studio 2015

The new Build System with Visual Studio 2015

The new builds are script based and web based. To build Windows, Azure, and other Visual Studio solutions, you need at least one Windows build agent. This agent can be a part of an agent pool which can be created by going to *Administer Server tab* > selecting *Collection* > and later Build tab from it. A new pool can be created for registering agents. Pools are available to run builds for all collections. The collection name can be specified while configuring the agent.

It is possible to manage agent pools (a groups of resources or agents), download and configure build agents by selecting the *administer account* from Web Access. Make sure the path for download is not too long.

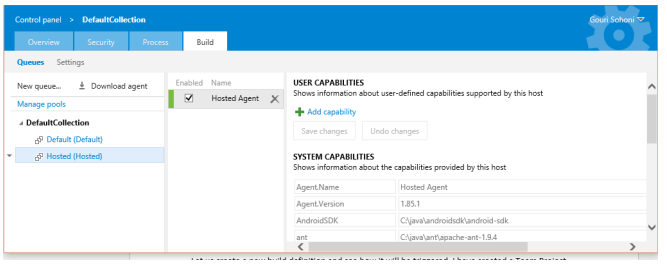


Figure 2: Manage Pools via Web Access

We can configure agents for on-premises Team Foundation Server, as well as for VSO. Figure 2 shows the hosted agent is configured. There is a link for Download Agent. Once the agent is downloaded, configure it by starting PowerShell as Administrator. In the downloaded folder, you will see a PowserShell script file named **ConfigureAgent.ps1.** Start this script from PowerShell command prompt. It will ask for various configuration related questions like name of agent (default), url for Team foundation Server, name of the pool (default) and how the agent should be running. We can run build agent as an interactive process or as a service. We can create pool as default or hosted, depending upon if we are using VSO or on-premises TFS.

If I select Manage pool, I get two groups - one for Agent Pool Administrators and other for Agent Pool Service Accounts. Administrators operate as a role, if any user is added to the Admin group, he/she can add or remove agents. Service accounts is to allow for agents to connect to the pool.

There are some restriction for using hosted pool (with VSO) though. Following are the scenarios where hosted pool will not be available:

- You are running XAML builds

- You need multiple builds to run for your account

- Build process is a long time process (takes almost more than an hour)

- Build process uses lot of disk space on build server (even more than 10 GB)

- You want to run build process in interactive mode

We can configure XAML build processes which are not migrated or want to keep as it is. For this, we have to use the 'XAML Build Configuration' tab shown in Figure 3 at the time of configuring Team Foundation Server, as we used to do in earlier versions. We can configure the build service to start automatically if required.

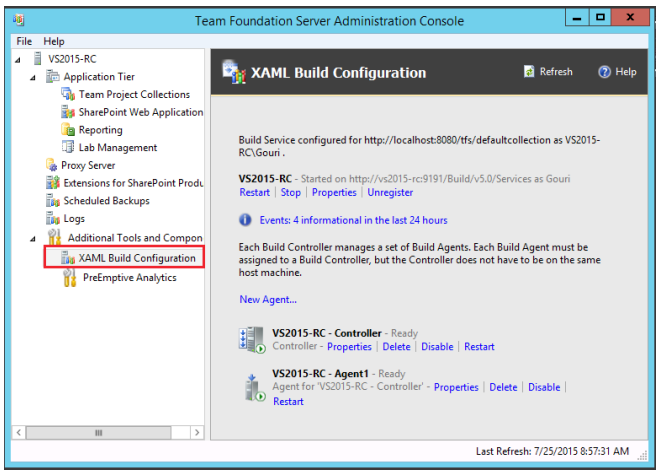


Figure 3: TFS Configuration wizard – Configure XAML Build

Creating a new Build Definition

Let us create a new build definition and see how it will be triggered. I have created a Team Project based on Scrum Process template and added it to TFVC (Team Foundation Version Control) for this walkthrough. Here are the steps involved.

1. Click on New Build Definition option from Visual Studio 2015 and you will get following screen as shown in Figure 4. It automatically takes you to Web access

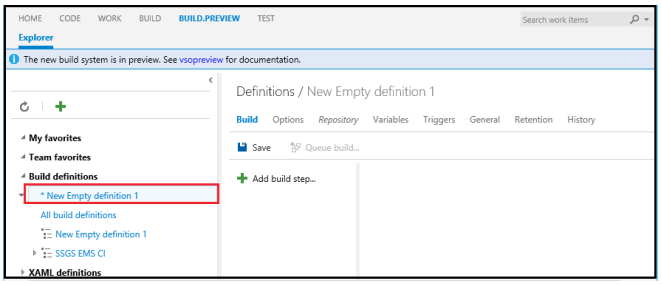


Figure 4: Create Build via Web Access

2. You can create build for Windows, iOS, Ant, Maven or on Linux also as step(s). Add MSBuild step and provide the solution to be built. With Visual Studio Online, definition templates are also available. You can create your own templates too.

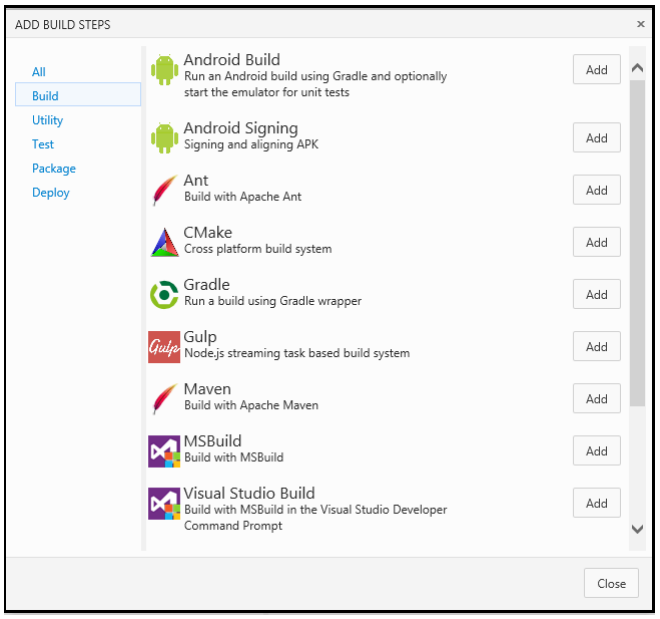


Figure 5: Add steps to Build

The Visual Studio Build task is added and the project to build is provided as shown in Figure 6.

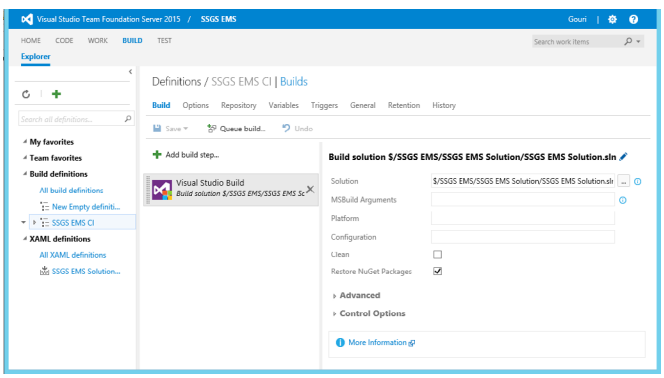


Figure 6: Visual Studio Build Step with solution added to built

3. The build can be edited and customized by using

script languages like batch script, PowerShell script or passing command line arguments.

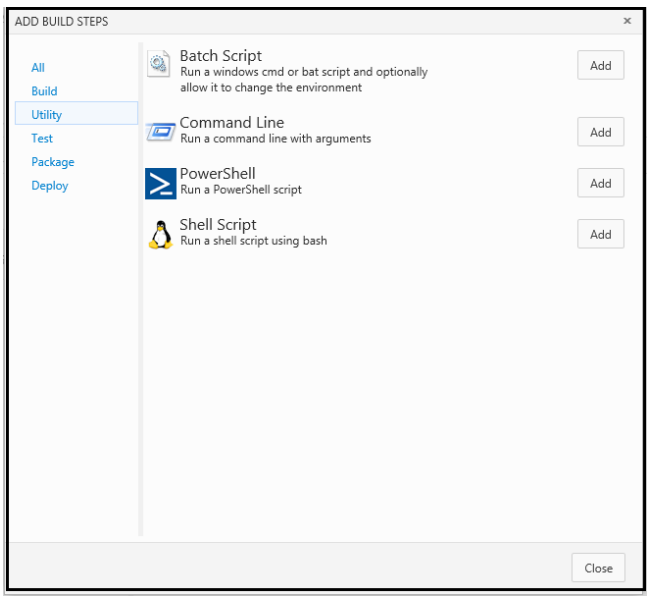


Figure 7: Build Definition – Run Utility for customizing

4. We can add Build Verification Tests using Visual Studio Test or Xamarin Test Cloud

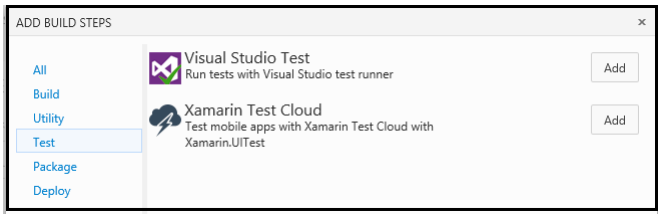


Figure 8: Build Definition –Adding Steps for Testing

If we select Visual Studio Online, we get more options for test tab. Note that we can run load testing as it is now supported with VSO.

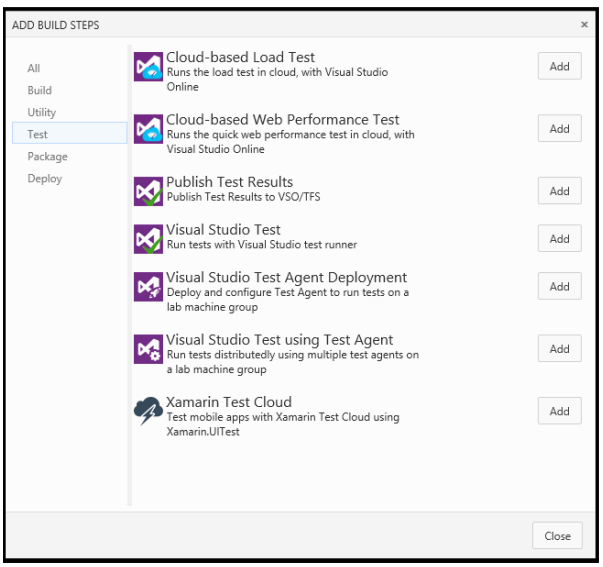


Figure 9: Build Definition – Adding Test Steps using VSO

5. Now provide deployment options like Azure Cloud Service or Azure Web site

Figure 10: Build Definition – Add Deployment Steps

6. MultiConfiguration option from Options tab shown in Figure 11 helps in building selected solutions multiple times. We can add comma separated list of configuration variables here. These variables are listed in the variables tab. You can find a list of predefines variables at this link. If we click on the Parallel check box, it will have all the combinations provided in parallel. The pre-requisite is that there should be multiple agents available to run. If the variables are declared, we need to set the values for them. We can even provide multiple values to variables by separating with comma.

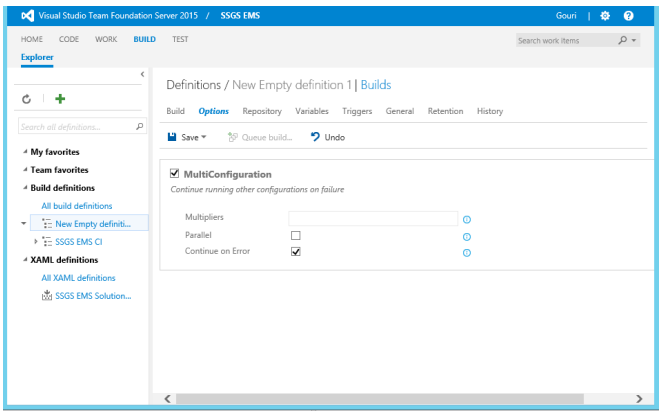


Figure 11: Build Definition – Provide multiple configuration facility

7. The next tab is for version control repository. You can choose TFVS (Team Foundation Version Control) or Git depending upon the project to build.

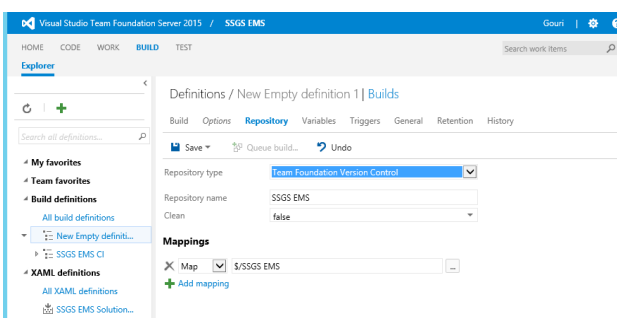


Figure 12: Build Definition – Adding repository from Source Control

8. There are two triggers available - Continuous Integration and Scheduled.

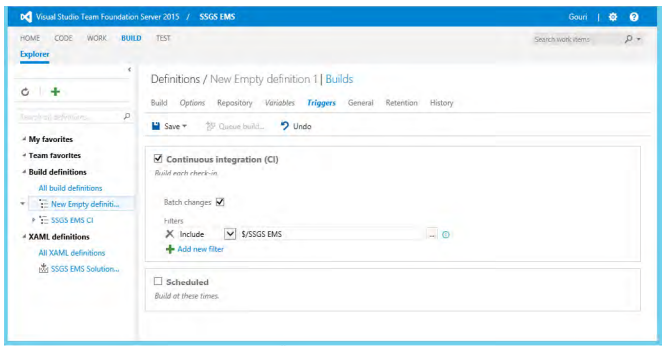


Figure 13: Build Definition – Provide Trigger for Build

9. If we want to add more steps we just have to go back to Build tab and add the required one. The build definition can be saved as a draft which can be edited anytime later.

10. Retention tab is to decide how long the builds will be retained. These settings can be changed by choosing the option to administer server, select Collection, select Build and finally select Settings tab. History will provide options as Save, Queue Build, Undo and Diff. We can select two builds and click on Diff tab which will provide us with the differences between the two builds(similar to diff tool for version control)

11. Once the build is created and saved, it will be automatically triggered with next check-in because of Continuous Integration. The existing build definition can also be saved as a template. Such templates can be used as base for new definitions. The triggered build will be executed by default agent or hosted depending upon if we are using on-premises TFS or VSO.

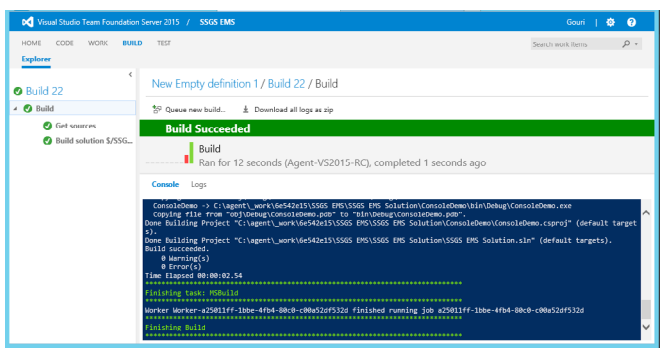


Figure 14: Successful completion of Build

The build is successfully completed.

We can add tags to the build so that we can filter the builds later. We can also find out the changeset which triggered the build as well as retain any build indefinitely if needed. At the time of build execution the console will show us live results as shown in Figure 14.

12. We can add steps for testing with Visual Studio and also configure Code Coverage results. Following build covers Test results as well as Code Coverage information.

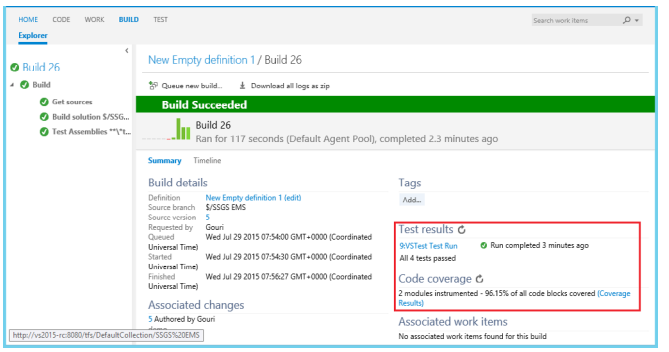


Figure 15: Build Completion with Test Results and Code Coverage analysed

Wrap up

Here's a wrap up of some important features of working with Visual Studio 2015 Build

- It gives a Web User Interface. Even if we start creating a build definition from Visual Studio, we end up working with Web UI.

- Customization is simple. We don't need to learn Windows workflow anymore.

- Cross platform builds are supported.

- With previous version of build, the build controller was configured to build service which in term was connected to a collection. Now we can create pools with agents.

- You can view complete logs. These logs can also be downloaded as zip file.

- If we have enabled Multiconfiguration and also have two build agents configured, we will be able to

see parallel builds.

- If we select any task on the left hand side of the explorer, we can see build summary for that task (e.g. Get Sources, Build Solution, Test Assemblies etc.)

- Facilities like running Tests as Build Verification Tests or finding Code Coverage information are still available

The new builds are web- and script-based, and are highly customizable. They leave behind many of the problems and limitations of the XAML builds. Try it out and see the difference for yourself ■

• • • • • • •

About the Authors

gouri sohoni

***Gouri Sohoni**, is a Visual Studio ALM MVP and a Microsoft Certified Trainer since 2005.*

Check out her articles on TFS and VS ALM at bit.ly/dncm-auth-gsoh

Microsoft® MVP Most Valuable Professional

Using Bootstrap, jQuery and Hello.js to integrate Social Media in your Websites

The web is fundamentally about people and the huge popularity of Social Media has echoed this notion. Social media is one of the must go-to places for everybody who surfs online. At times, we can't decide what to surf on the internet unless we check one of the social media sites. Most of the corporations use social media to post their news, updates and even for important announcements. Social media integration on websites also gives the viewer an impression of the social presence of the company or, the individual.

More consumers are connected than ever before, and if you are not engaging with them via social media, you are wasting an opportunity.

Every social media site like Twitter, Facebook etc. provides APIs using which, one can view or post updates via an app or, another site. The app or, site using social media has to be authenticated and should have enough rights to do so. For this, the site should be authenticated via OAuth and use the OAuth token for each interaction. Handling the task of authentication and making sure to send the security details on each interaction is made possible via libraries like hello.js .

In this article, we will see how hello.js can be used to fetch feeds from some of the most widely used social media sites and display them on a webpage using Bootstrap and jQuery.

Understanding OAuth

If you haven't heard of OAuth or OpenID, you have most certainly used it in day to day life. Have you provided your google username/password and logged in to StackOverflow OR used your WordPress. account to post comments on someone's blog OR used third party Twitter clients whom you have given permission to use your Twitter account OR used Facebook to post comments on some tech blog site? If answer to any of the above is a YES, you have seen OAuth or OpenID in action. As defined on Techopedia – "*OAuth is an open-source protocol that enables users to share their data and resources stored on one site with another site, under a secure authorization scheme, based on a token-based authorization mechanism*".

I am taking an excerpt from an article written by Sumit Maitra where he explains OAuth accurately.

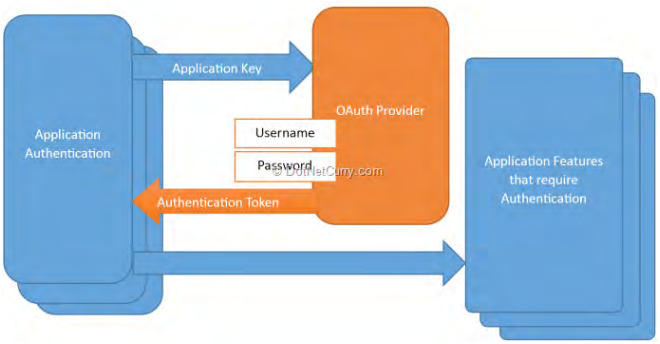
Basic premise of OAuth is, instead of having to remember a different username and password for every application we visit on the internet, we have a dedicated Authentication provider with whom we can register our user name and password. Applications (aka Relying Party) redirect us to this provider for authentication and after a successful authentication; the provider passes back a token to the replying party who initiated the request. The target Application then uses the token to verify the identity of the person and provide further access to Application Features.

OAuth was initiated by Twitter when they were working on building their OpenID implementation for Twitter API. Their need was more towards controlling what features to make available based on Authentication Types offered at the Provider and those sought by the relying party. For example when a Relying party signs up for Twitter Authentication access, they can request for a Read-Only connection (this gives them read permissions to a user's stream), a Read/Write connection (this gives them read and write permissions on a user's stream) and a Read/Write with Direct Messages connection (giving them maximum possible access to user's data). Thus, when a user authenticates with Twitter over OAuth, they are told exactly what kind of access they are providing.

On the other hand, OpenID simply ensures that you are who you claim to be by verifying your username and password. It has no way of controlling feature access.

In even simpler terms and for the sake of explanation, OpenID is about Authentication, OAuth is about Authentication and Authorization (for features at the Provider).

Shown here is an architecture of External Authentication using OAuth 2.0 –



The diagram above demonstrates the Authentication process for an OAuth provider. The overall workflow is as follows:

1. Application (Relying Party) registers with OAuth provider and obtains an Application specific key (secret). This is a one-time process done offline (not shown in the image).

2. Next the Application (Relying Party) has to pass the Application key to the provider every time it intends to authenticate someone.

3. User of the Relying Party needs to be registered with the Provider (again a one-time offline process)

4. Every authentication request directs the user to the Auth Provider's site. The user enters the Username and Password that they obtained by

creating an account with the provider. The Provider verifies the account credentials, then it matches the Application Key and provides a token back to the Relying Party with which only the Authorized actions can be performed.

For more info see: http://oauth.net

Connect to Twitter, Facebook and Instagram - The implementation

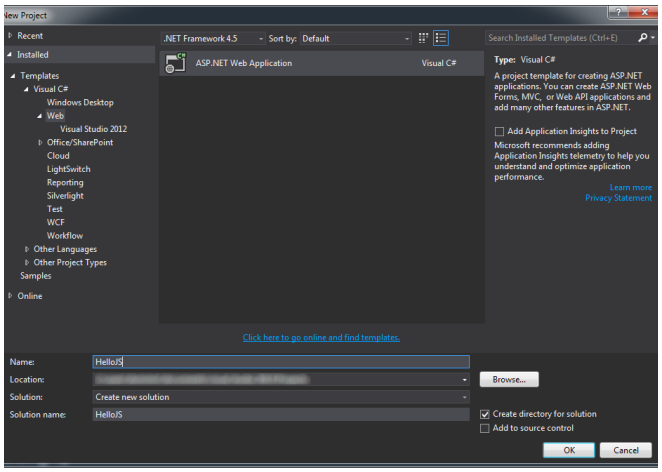
With OAuth knowledge under our belt, we will write code that allows us to connect to three social media sites - Twitter, Facebook and Instagram using only JavaScript .

To help us with this objective, we'll use a very useful library from Andrew Dodson called Hello.js (http://adodson.com/hello.js/).

> Hello.js is a client-side JavaScript SDK for authenticating with OAuth2 (and OAuth1 with a oauth proxy) web services and querying their REST APIs. HelloJS standardizes paths and responses to common APIs like Google Data Services, Facebook Graph and Windows Live Connect. It's modular, so that list is growing. No more spaghetti code!

Hello.js is a library that saves us from all the complex stuff behind an OAuth authentication, presenting us with a simple model and some easy to use API's.

Let's create a website using the Free Visual Studio Community Edition or Visual Studio Code editor and use Hello.js in it. Open Visual Studio and create a new empty website.



Visual Studio automatically provides a local server to host our website. This is a necessary step because, as stated previously, OAuth is a dialogue between two servers.

Now that our local server is setup, we can create our OAuth clients on Twitter, Facebook and Instagram. Normally these actions can be performed in the "Developer" section of many social media sites.

At the end of the process, we would have registered a trusted application with a Client ID value, necessary for OAuth connections.

Generating Keys from Social Media sites

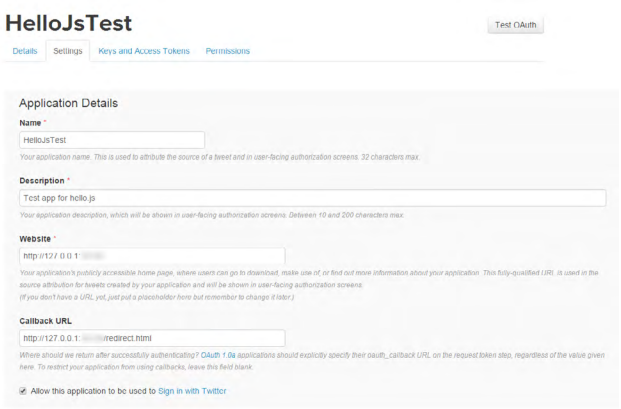
For Twitter:

1. Go to http://dev.twitter.com

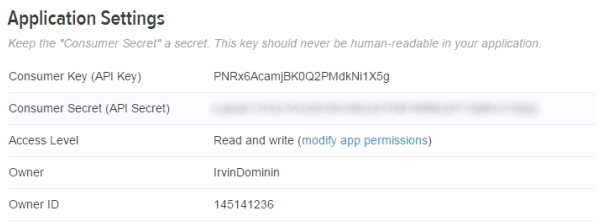
2. Login with your Twitter account and select "My applications" in your avatar menu

3. Hit "Create a new application"

Add some basic information like "Application Name" and "Description" and some important data like the "Website" and "Callback URL". Fill these fields with our website data as shown in the following figure:



At the end of the registration process, Twitter will release an API Key and an API Secret for your app:



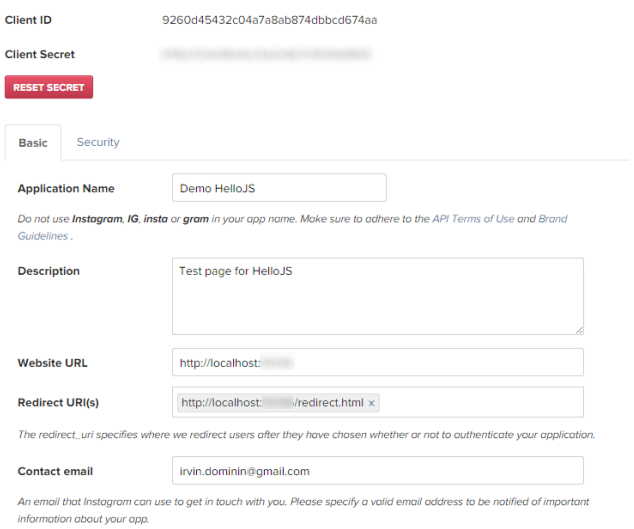
For Instagram:

1. Go to https://instagram.com/developer

2. Login with your Instagram account and hit "Register Your Application"

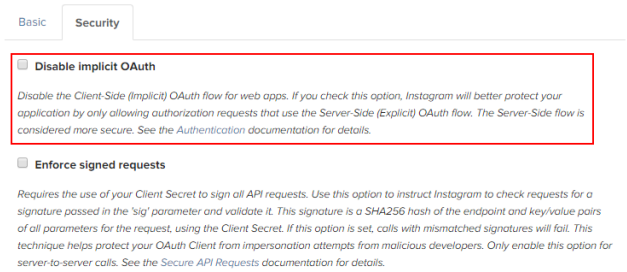
3. Hit "Register a New Client"

Here add some basic information like "Application Name" and "Description" and some important data like the "Website URL" and "Redirect URI(s)". Fill these details with our website data as shown here:



The Redirect URI(s) field points to a redirect page.

We will create the redirect.html page shortly. Don't forget to "Disable Implicit OAuth" in the Security tab in order for the app to work correctly in our scenario:



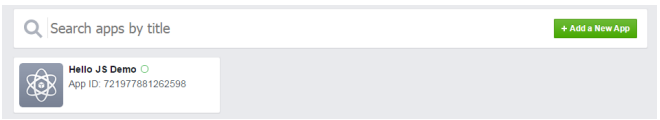
At the end of the registration process, Instagram will release an API Key and an API Secret for your app.

For Facebook:

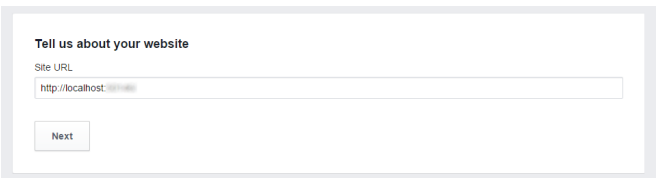
1. Go to http://developer.facebook.com

2. Login with your Facebook account and select "My applications" on your avatar menu

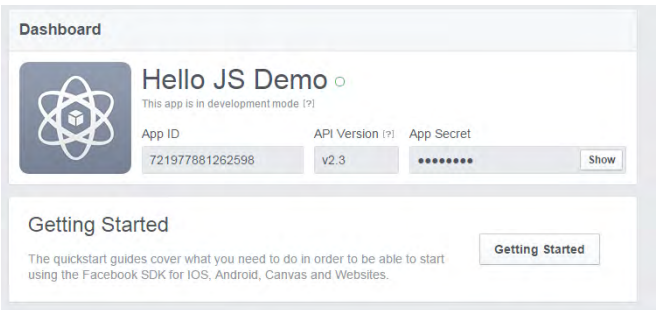
3. Hit "Add a New App"



Here add some basic information like "Application Name" and "Description" and some important data like the "Site URL". Fill these details with our website data as follows:



At the end of the registration process, Facebook will release an API Key and an API Secret for your app:



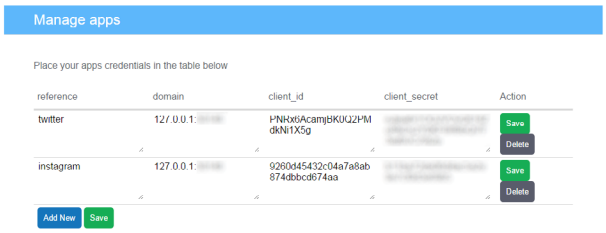
For OAuth1 or OAuth2 authentication with explicit grant services like Twitter, we need to use an OAuth Proxy service exposed by Hello.js.

For Hello.js proxy registration:

1. Go to https://auth-server.herokuapp.com

2. Login using a supported provider

3. Add your app to the "Managed apps" list



Once our setup is now complete, it is time to see some code.

The JavaScript code

In order to write code that can be reused in many situations, we will write a jQuery plugin. For those who are not familiar with jQuery plugin authoring and usage, you can refer to these articles on DotNetCurry http://www.dotnetcurry.com/jquery/1069/authoring-jquery-plugins and http://www.dotnetcurry.com/jquery/1109/jquery-plugin-for-running-counter.

Data displaying

Let us create a new page called **Plugin.html** that will include all necessary plugins and add references to some JavaScript and CSS code. In the <head> section, add the following reference:

```
<script type="text/javascript"
src="http://code.jquery.com/jquery-
1.9.1.js"></script>

<script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/
libs/hellojs/1.6.0/hello.all.min.js">
</script>

<script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/
libs/masonry/3.3.1/masonry.pkgd.min.
js"></script>

<script type="text/javascript"
src="jQuery.socialmedia.js"></script>

<link rel="stylesheet" type="text/css"
href="http://maxcdn.bootstrapcdn.com/
bootstrap/3.2.0/css/bootstrap.min.css">

<link rel="stylesheet" type="text/css"
href="https://maxcdn.bootstrapcdn.com/
font-awesome/4.3.0/css/font-awesome.min.
css">

<link href="jQuery.socialmedia.css"
rel="stylesheet" />
```

And add an empty div and two buttons on the page as shown here. The buttons and page are styled using Bootstrap:

```
<body>
 <div class="container-fluid">
  <h1>My social media accounts!</h1>
    <button class="btn btn-primary"
    id="connect">Connect to socials
    </button>

    <button class="btn btn-primary"
    id="fetch">Get data!</button>

    <div id="mySocialMedia"
    style="margin-top: 20px"></div>
 </div>
</body>
```

Plugin initialization

The following script runs in the document.ready() event on the "#connect" button click event handler and fills the div "mySocialMedia" with user profile information coming from Twitter, Instagram and Facebook.

```
$("#mySocialMedia").socialmedia({
 twitter: {
  enabled: true,
  key: TWITTER_CLIENT_ID
 },
 instagram: {
  enabled: true,
  key: INSTAGRAM_CLIENT_ID
 },
 facebook: {
```

```
  enabled: true,
  key: FACEBOOK_CLIENT_ID
 }
});
```

In the plugin call, we must change TWITTER_CLIENT_ID, INSTAGRAM_CLIENT_ID and FACEBOOK_CLIENT_ID keys, with the API keys (NOT with the secret key), as provided in the registrations we performed on individual social media sites.

The Redirection page

Now add a new page called redirect.html. This page is necessary for OAuth in order to work correctly, and is the same page we have defined previously in the Twitter and Instagram app registrations.

The redirect.html page is handled by Hello.js and is purely to indicate redirection.

```
<html>
<head>
 <title>Hello, redirecting...</title>
 <meta name="viewport"
 content="width=device-width, initial-
 scale=1.0, user-scalable=yes" />
</head>
<body>

 <div class="loading"><span>&bull;
 </span><span>&bull;</span>
 <span>&bull;</span></div>

 <h2>Please close this window to
 continue.</h2>
 <script>
  window.onbeforeunload = function () {
   document.getElementsByTagName('h2')
   [0].innerHTML = "Redirecting,
   please wait";
  }
 </script>

 <script type="text/javascript"
  src="https://cdnjs.cloudflare.com/ajax/
  libs/hellojs/1.6.0/hello.all.min.js">
 </script>
</body>
</html>
```

HelloJs initialization and Socials login

Let's see the plugin code - the core of this article. Inside the Init() function of the plugin, we can see the Hello.js integration. The code based on the options passed, can activate and login to all supported social networks (Instagram, Twitter and Facebook) and display login information.

```
// If twitter is enabled initalize hello
js for it
if (_opts.twitter.enabled) {
 hello.init({
  twitter: _opts.twitter.key
 },
  {
   redirect_uri: 'redirect.html', //
   Redirected page
   oauth_proxy: 'https://auth-server.
   herokuapp.com/proxy' // Since Twitter
   use OAuth2 with Explicit Grant we
   need to use hello.js proxy
  });

  // Twitter connection handler
  setTimeout(function () {
  connectToTwitter($twitterEl); }, 2000);
 }

 // If instagram is enabled initalize
 hello js for it
 if (_opts.instagram.enabled) {
  hello.init({
   instagram: _opts.instagram.key
  },
   {
    redirect_uri: 'redirect.html',
 });

 // Instagram connection handler
 setTimeout(function () {
 connectToInstagram($instagramEl); },
 2000);
}

 // If facebook is enabled initalize
 hello js for it
 if (_opts.facebook.enabled) {
  hello.init({
   facebook: _opts.facebook.key
  },
   {
    redirect_uri: 'redirect.html',
 });
```

```
  // Facebook connection handler
  setTimeout(function () {
  connectToFacebook($facebookEl); },
  2000);
 }
}
```

Now that we are logged in to our favorite social media sites, the next step is to fetch our feeds from these socials!

Social Data fetching

The script we will see shortly runs in document.ready() event on the "#fetch" button click event handler and fills the div "mySocialMedia" with data fetched from Twitter, Instagram and Facebook.

The fetch method uses Hello.js in order to simplify all the OAuth request to each social site, with some very simple API's.

For demo purposes, all the data fetched from the socials are added in an array, shuffled and displayed in a responsive grid system using Masonry (http://masonry.desandro.com/) from David DeSandro.

Let's see the plugin code. Inside the fetch() function of the plugin, we can see the Hello.js and Masonry integration. Since each response from the socials is asynchronous, we will use jQuery's deferred API. Deferred is an important concept and if you are not familiar with it, check: https://api.jquery.com/jquery.deferred/ and http://www.dotnetcurry.com/jquery/1022/jquery-ajax-deferred-promises

The code sends request to each social and concats and shuffles the response inside an array. The social data fetch code shown here is only for Twitter, however the complete code will contain details for all the three social media sites we are connecting to.

First we will create three Deferred objects and on completion, call the rendering function inside the main div.

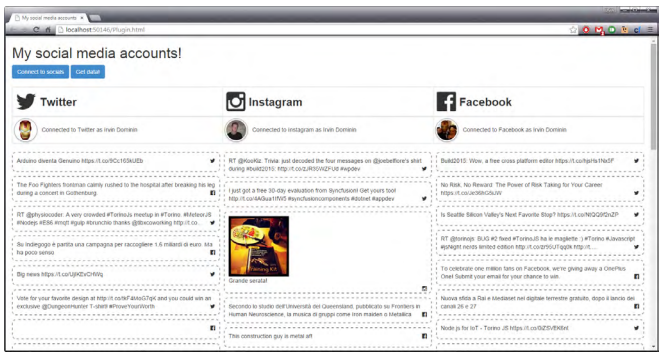
```
// Deferred objects
var d1 = $.Deferred();
var d2 = $.Deferred();
var d3 = $.Deferred();

// Waiting for the deferred completion
$.when(d1, d2, d3).done(function (v1,
v2, v3) {
 // Build one concatenated array with
 the result of each data fetch result

 fecthedData = fecthedData.concat(v1);
 fecthedData = fecthedData.concat(v2);
 fecthedData = fecthedData.concat(v3);

 // Schuffle the array result
 fecthedData = shuffleArray(fecthedData);

 // Render the mixed fetched data
 renderFetchedData(_$this, fecthedData);
});

 fetchTwitterData(_opts.twitter, d1);
 fetchInstagramData(_opts.instagram,
d2);
 fetchFacebookData(_opts.facebook, d3);
}
```

Since we are logged in to Twitter in the Init() code, if twitter integration is enabled in the plugin initialization, we get its instance and fetch data from twitter. The fetch is done using Hello.js that simplifies this operation. The core functionality of the app requests twitter for the 50 latest tweets (or a number you chose) and loads them in the "tweet" array. When the data fetch is completed, the deferred object "resolve" method is called and the operation continues.

```
function fetchTwitterData(o, d) {

 if (!o.enabled) {
  d.resolve([]);
  return;
 }

 // Twitter instance
 var twitter = hello('twitter');

 twitter.api('twitter:/me/share', {
  limit: o.maxElements }, function (r) {
   var tweets = [];
   for (var i = 0; i < r.data.length;
   i++) {
    var o = r.data[i];
    tweets.push({ social: 'twitter',
    text: o.text });
   };
  d.resolve(tweets);
 });
}
```

Result

Run the page and hit "Connect to socials" and then click on "Get data!" buttons. We will see our social profile images and then our social media data (tweets, facebook posts and instagram images) displayed.



Conclusion

Social media integration is one of the key aspects for several web sites. A number of websites display their media feeds on their pages to show the visitors their active presence on the social media. Hello.js makes this process easier by providing an easy to use API. This article covered a simple example of the integration and I hope you found it useful enough to use it on your own websites ■

Download the entire source code from GitHub at
bit.ly/dncm20-hellojssocialmedia

• • • • • • •

About the Authors

Irvin Dominin

Irvin Dominin is currently working as lead of a technical team in SISTEMI S.p.A. (Turin, Italy) on .NET, jQuery and windows Projects. He is an active member on StackOverflow http://stackoverflow.com/users/975520/irvin-dominin). You can reach him at: irvin.dominin@gmail.com or on LinkedIn: https://it.linkedin.com/in/irvindominin

USING NEW XAML TOOLS IN VISUAL STUDIO 2015

With an increase in XAML based apps for Windows Phone, Windows 8+ and Universal apps, developers have been demanding new features for XAML in the areas of memory and CPU profiling, UI debugging, seamless integration between Visual Studio and Blend. The latest release of Visual Studio 2015 provides some new XAML development features for applications. In this article, we will take an overview of some of these new features.

Seamless Integration with Blend

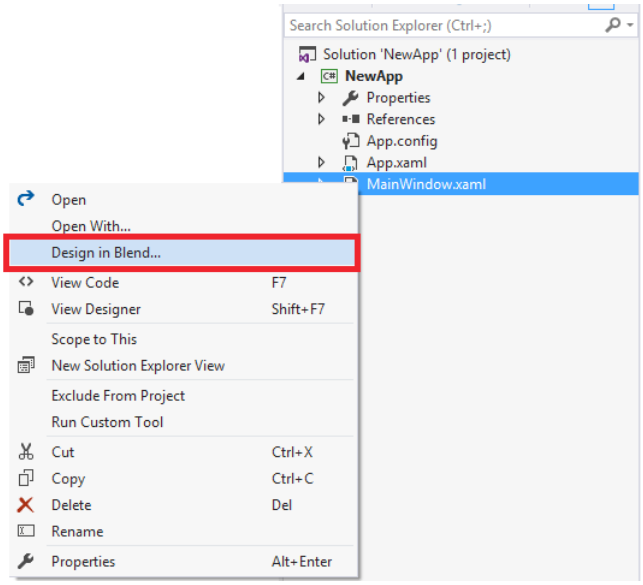
Application development in XAML needs efforts from both UI Designer and Developer. In the development process, the UI Designer uses Blend to manage XAML design using design templates, styles, animations etc. Similarly the UI Developer uses Visual Studio to add functionality to the design. Many-a-times it is possible that the UI Designer and Developer work on the same XAML file. In this case, the file updated by the designer or developer should be reloaded by the IDE (Blend and Visual Studio) each time. Visual Studio 2015 contains new settings that allows the integration of the two IDEs to be seamless. Let's explore this setting.

Step 1: Open Visual Studio 2015 and create a new WPF Application. In this project, open MainWindow.xaml and add a button to it. Here's the XAML code:

```
<Button Name="btn" Content="Click"
  Height="50"
  Width="230"
  Click="btn_Click"
  Background="Brown"></Button>
</Grid>
```

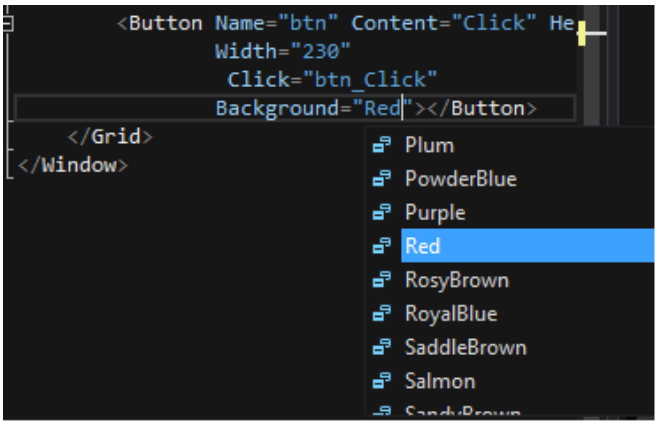
To update the XAML in Blend, right-click on the MainWindow.xaml and select **Design in Blend**

option as shown in the following image:



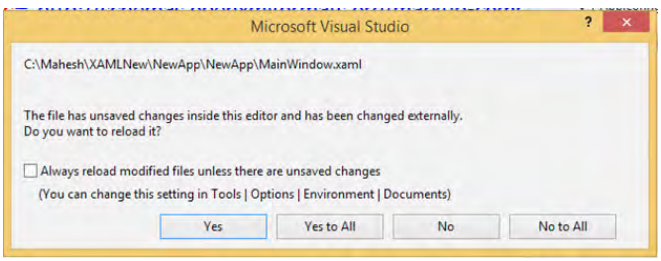
This will open the file in Blend. This blend version is more similar to the Solution Explorer window in Visual Studio.

Step 2: In Blend, change the **Background** property of the button to *Red* as shown here. (Observe the intellisense).



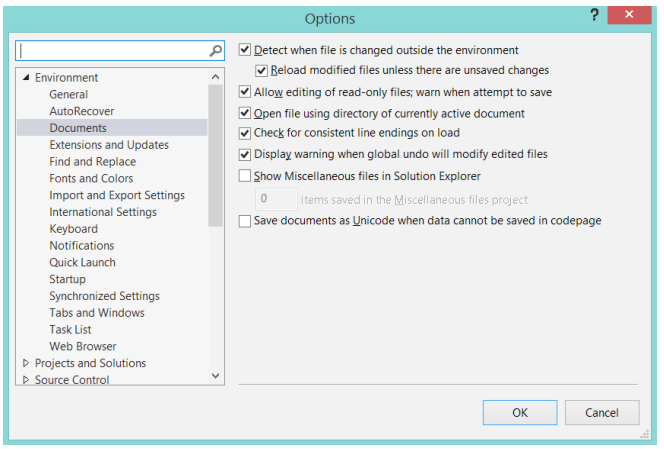
Save the file.

Step 3: Visit the project again in Visual Studio, and the following window will be displayed:



The above window notifies that the file is updated externally. To reload the file with the external changes, we need to click on **Yes** or on the **Yes to All** button. Once clicked, you will observe that the updates made in Blend will be reflected in Visual Studio. In Visual Studio the Background property will be updated from *Brown* to *Red*. We can manage this reload with seamless integration with the following settings.

In Visual Studio 2015, from **Tools |Options |Environment |Documents** select the checkbox as shown in the following image:



The CheckBox **Reload modified files unless there are unsaved changes** configuration will load changes made in the XAML file outside Visual Studio editor. Similar changes can be configured in Blend using **Tools |Options |Environment |Documents**. With this new feature, we can implement seamless integration between Visual Studio and Blend for efficiently managing XAML updates by the Designer and Developer.

Using Live Visual Tree for UI Debugging

In the process of DataBinding, new XAML elements may be added dynamically. To detect the UI elements added dynamically, we need a smart tool. In XAML based applications (e.g. WPF) the arrangement of XAML elements with its dependency properties is known as **Visual Tree**. In the VS 2015 release, a **Live Visual Tree** tool is provided. This tool helps to inspect the Visual Tree of the running WPF application and properties of element in the Visual Tree.

The Live Visual tree can be used for the following purposes:

- to see the visual tree with dynamically generated UI Elements based on the DataBinding, Styles, etc.

- to identify the critical sections of the generated Visual Tree which result in performance degradation.

Step 1: Open MainWindow.xaml and update the XAML as shown in the following code:

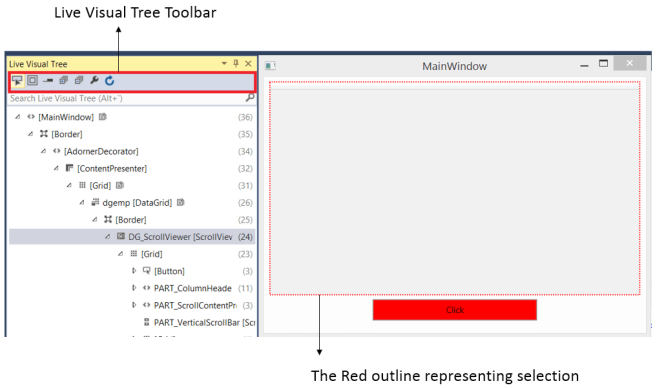
```
<Grid Height="346" Width="520">
  <Grid.RowDefinitions>
    <RowDefinition Height="300">
    </RowDefinition>
    <RowDefinition Height="40">
    </RowDefinition>
  </Grid.RowDefinitions>
  <DataGrid  Name="dgemp"
  AutoGenerateColumns="True" Grid.
  Row="0"/>
  <Button Name="btn" Content="Click"
  Height="30" Width="230"
  Click="btn_ Click" Grid.Row="1"
  Background="Red"></Button>
</Grid>
```

Step 2: In the Code behind, add the following C# Code:

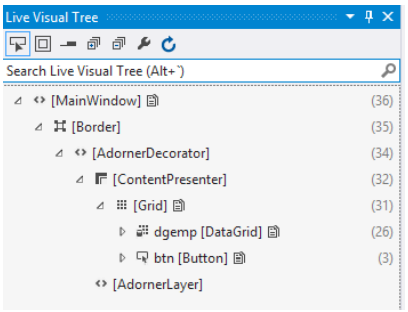
```
public partial class MainWindow : Window
{
 public MainWindow()
 {
  InitializeComponent();
 }

 private void btn_Click(object sender,
 RoutedEventArgs e)
 {
  //this.Background = new
  SolidColorBrush(Colors.RoyalBlue);
  dgemp.ItemsSource = new
  EmployeeList();
 }
}

public class Employee
{
 public int EmpNo { get; set; }
 public string EmpName { get; set; }
}

public class EmployeeList :
List<Employee>
{
 public EmployeeList()
 {
  Add(new Employee() { EmpNo = 1,
  EmpName="A"});
  Add(new Employee() { EmpNo = 2,
  EmpName = "B" });
  Add(new Employee() { EmpNo = 3,
  EmpName = "C" });
  Add(new Employee() { EmpNo = 4,
  EmpName = "D" });
  Add(new Employee() { EmpNo = 5,
  EmpName = "E" });
 }
}
```

The above code defines the Employee class with Employee properties and EmployeeList class containing Employee Data. On the click event of the button, the EmployeeList is passed to ItemsSource property of the DataGrid.

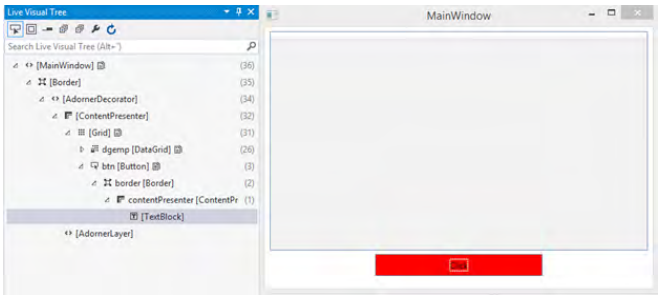
Step 3: Run the Application. A WPF window will be displayed with Live Visual Tree panel as shown in the following diagram:



The Live Visual Tree has an Icon tool bar which provides options like **Enable Selection in Running Application** and **Preview Selection**. The above image shows the non-expanded Visual Tree. After expanding the Visual Tree, the panel gets displayed as shown in the following image:

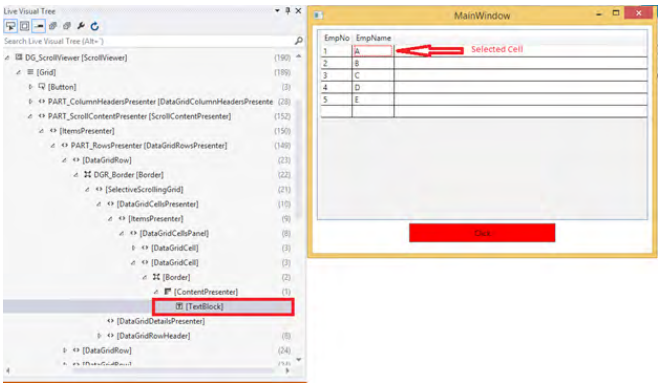


This shows DataGrid and a Button. Selecting the Button using the Mouse will select the Content property of the Button as shown in the following image:



The ContentPresenter contains TextBlock containing **Click** text.

Step 4: Click on the **Enable Selection in running application**. This will clear the Button selection in the Live Visual Tree. Click on the button and the DataGrid will show Employee Data. Click in the **Enable Selection in Running application** and **Preview Selection** from the Live Visual Tree toolbar and select a row in the DataGrid. The Dynamically generated Visual Tree will be displayed as shown in the following Image



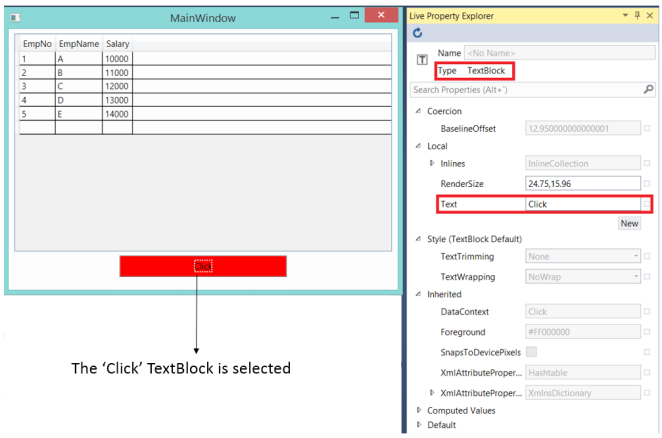
The above diagram shows the dynamically generated Visual Tree for the DataGrid. The ItsmsSource property of the DataGrid generates DataGridRow, which further contains DataGridCell, which in turn contains TextBlock. As we keep changing the selection in the DataGrid, the Live Visual Tree helps to debug the UI selection.

Hence this new tool helps developers to inspect the UI elements generated.

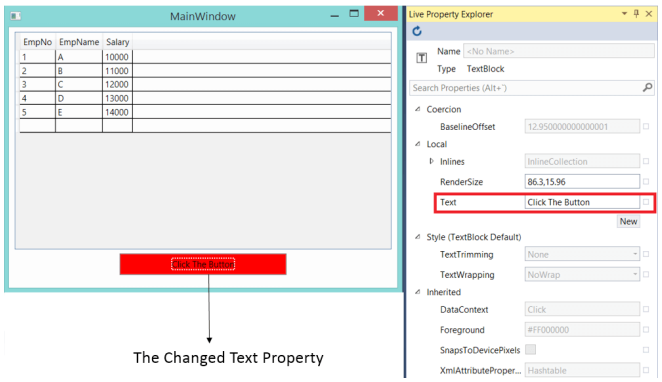
Live Property Explorer

Along with the Live Visual Tree, we have a Live Property Explorer tool as well. This tool shows the property set applied on the selected UI element. This also allows us to change some of the property values of selected element, at run time.

Step 1: Select the button on the running application, the property explorer will be displayed as shown in the following image:



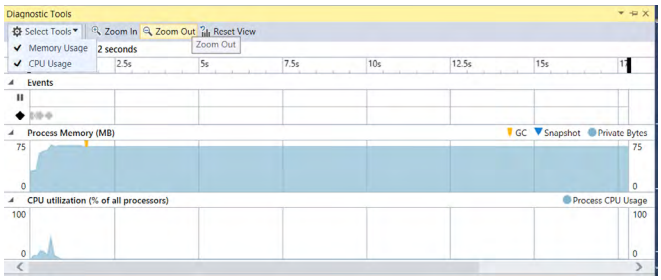
The Click Text is selected for the button, this shows the Live Property explorer for the TextBlock. Here we can change the **Text** property of the TextBlock while the application is running, as shown in the following image:



As you saw, the Live property explorer tool helps to update some properties while running the application. These changed properties can also be directly applied to the application.

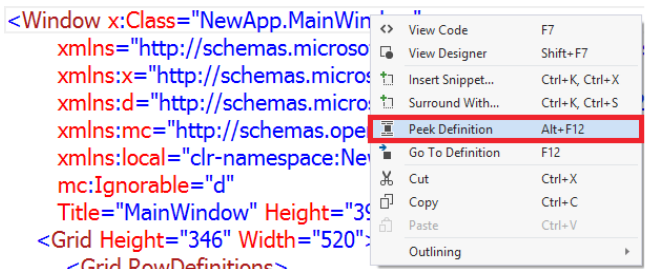
Diagnostic Tool

A new feature provided for XAML based application with this new release of Visual Studio 2015 is the **Diagnostic Tools**. This tool allows developers to check the Memory and CPU utilization for the XAML application. This tool can be enabled using **Debug| Show Diagnostic Tool**. Here we can select **Memory Usage** and **CPU Usage**. Run the application, and the CPU and Memory utilization can be seen as shown in the following diagram:

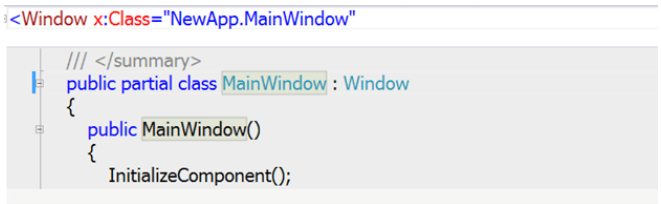


XAML Peak

In Visual Studio 2015, a new feature for developers called the **XAML Peek Definition** has been introduced. In the earlier versions of Visual Studio, we could peek into definition for classes, functions, etc. in C# and VB.NET language. In Visual Studio 2015 we can use peek definitions for XAML elements as well. In the following diagram, we can see that when we right-click on the **x:Class** attribute value e.g. **NewApp.MainWIndow** and select the **Peek Definition** option from the context menu, we can see the class definition.



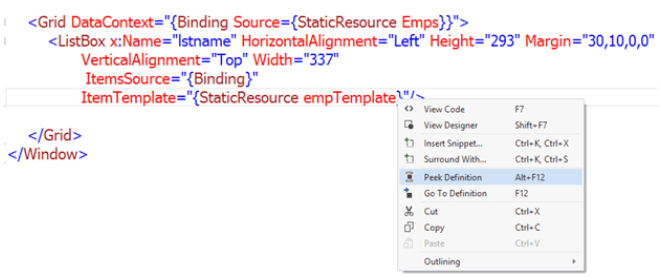
We can view the definition as shown in the following figure:



```
<Window x:Class="NewApp.MainWindow"
```

```
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
```

The advantages of this feature is that we can also use this to show styles/DataTemplate implementation for selected XAML elements as shown in the following diagram:

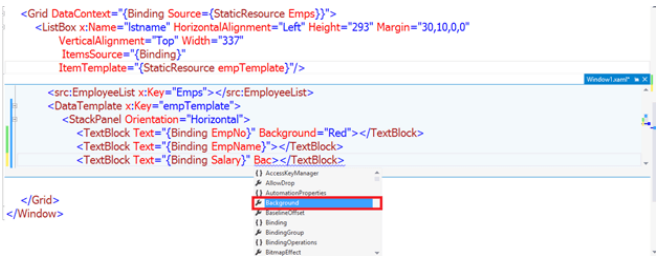


```
    <Grid DataContext="{Binding Source={StaticResource Emps}}">
        <ListBox x:Name="lstname" HorizontalAlignment="Left" Height="293" Margin="30,10,0,0"
                 VerticalAlignment="Top" Width="337"
                 ItemsSource="{Binding}"
                 ItemTemplate="{StaticResource empTemplate}"/>

    </Grid>
</Window>
```

In the above diagram, the definition of the **empTemplate** can be shown:

```
<Grid DataContext="{Binding Source={StaticResource Emps}}">
    <ListBox x:Name="lstname" HorizontalAlignment="Left" Height="293" Margin="30,10,0,0"
             VerticalAlignment="Top" Width="337"
             ItemsSource="{Binding}"
             ItemTemplate="{StaticResource empTemplate}"/>

    <src:EmployeeList x:Key="Emps"></src:EmployeeList>
    <DataTemplate x:Key="empTemplate">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding EmpNo}" Background="Red"></TextBlock>
            <TextBlock Text="{Binding EmpName}"></TextBlock>
            <TextBlock Text="{Binding Salary}"></TextBlock>
```

Now we can edit one of the TextBlock e.g. Background property of the Salary TextBlock as follows:



```
<Grid DataContext="{Binding Source={StaticResource Emps}}">
    <ListBox x:Name="lstname" HorizontalAlignment="Left" Height="293" Margin="30,10,0,0"
             VerticalAlignment="Top" Width="337"
             ItemsSource="{Binding}"
             ItemTemplate="{StaticResource empTemplate}"/>

    <src:EmployeeList x:Key="Emps"></src:EmployeeList>
    <DataTemplate x:Key="empTemplate">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding EmpNo}" Background="Red"></TextBlock>
            <TextBlock Text="{Binding EmpName}"></TextBlock>
            <TextBlock Text="{Binding Salary}" Bac></TextBlock>

</Grid>
</Window>
```

The changes made in the Peek Definition window can be seen in the actual DataTemplate definition. This is a very cool feature that excites the developer in me and hopefully yours too.

DataBinding Debugging

In Line-of-Business (LOB) application development in WPF, we implement databinding with XAML elements using the **Binding** class. We implement hierarchical DataBinding using **DataContext** property. In Visual Studio 2015 using the XAML extended feature, we can experience DataBinding debugging using **Live Property Explorer**.

Consider the following C# class:

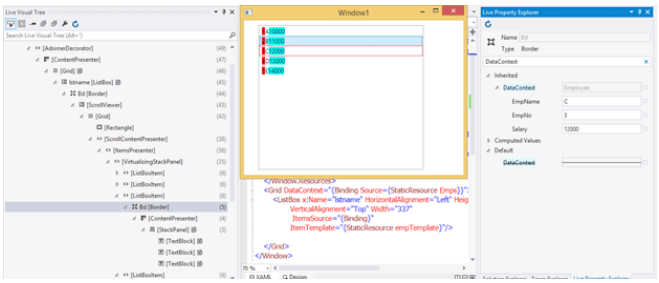
```
public class Employee
{
    public int EmpNo { get; set; }
    public string EmpName { get; set; }
    public int Salary { get; set; }
}

public class EmployeeList : ObservableCollection<Employee>
{
    public EmployeeList()
    {
        Add(new Employee() { EmpNo=1,EmpName="A",Salary=10000});
        Add(new Employee() { EmpNo = 2, EmpName = "B", Salary = 11000 });
        Add(new Employee() { EmpNo = 3, EmpName = "C", Salary = 12000 });
        Add(new Employee() { EmpNo = 4, EmpName = "D", Salary = 13000 });
        Add(new Employee() { EmpNo = 5, EmpName = "E", Salary = 14000 });
    }
}
```

Consider the following XAML with Databinding:

```
    xmlns:src="clr-namespace:NewApp"
    xmlns:local="clr-namespace:NewApp"
    mc:Ignorable="d"
    Title="Window1" Height="353.008" Width="464.662">
<Window.Resources>
    <src:EmployeeList x:Key="Emps"></src:EmployeeList>
    <DataTemplate x:Key="empTemplate">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding EmpNo}" Background="Red"></TextBlock>
            <TextBlock Text="{Binding EmpName}"></TextBlock>
            <TextBlock Text="{Binding Salary}" Background="Cyan"></TextBlock>
        </StackPanel>
    </DataTemplate>
</Window.Resources>
<Grid DataContext="{Binding Source={StaticResource Emps}}">
    <ListBox x:Name="lstname" HorizontalAlignment="Left" Height="293" Margin="30,10,0,0"
             VerticalAlignment="Top" Width="337"
             ItemsSource="{Binding}"
             ItemTemplate="{StaticResource empTemplate}"/>

</Grid>
```

The above XAML applies to the EmployeeList instantiated with Emps.

Run the application and enable the **Live Visual Tree** and **Live Property Explorer**. We can see the DataContext property in the Live Property Window. We can now experience the DataContext values by selecting the Employee record from the ListBox as shown in the following figure:



The above diagram shows the record selected from the ListBox (Red outline). On the right side of the diagram see the **Live Property Explorer** with the **DataContext** values. This is a cool feature for developers working on the LOB applications using XAML.

Defining Regions in XAML

While working with C# or VB.NET code in Visual Studio, we can make use of **Region-EndRegion** feature for defining groups of code for better manageability. In Visual Studio 2015, for long XAML markups, we can now define regions as shown in the following figure:


```
<!--#region  The List with Binding-->
<ListBox x:Name="lstname" HorizontalAlignment="Left" Height="293" Margin="30,10,0,0"
         VerticalAlignment="Top" Width="337"
         ItemsSource="{Binding}"
         ItemTemplate="{StaticResource empTemplate}"/>
<!--#endregion-->
```

The Region after collapse will be displayed as shown in the following diagram:

```
<Grid DataContext="{Binding Source={StaticResource Emps}}">
    The List with Binding
</Grid>
```

Although this is a really simple feature, imagine the pain of the developer who works on long and complex XAML code for maintenance and could not group it until now.

Conclusion

The new XAML tools provided in VS 2015 helps developers to effectively manage and work with XAML based applications in areas like UI Debugging and Performance ■

Download the entire source code from GitHub at
bit.ly/dncm20-xamlnewtools

About the Author

Mahesh Sabnis *is a Microsoft MVP in .NET. He is also a Microsoft Certified Trainer (MCT) since 2005 and has conducted various Corporate Training programs for .NET Technologies (all versions). Follow him on twitter @maheshdotnet*

Mahesh blogs regularly on .NET Server-side & other client-side Technologies at bit.ly/HsS2on