ISSUE 17 | MAR - APR 2015

DNCMAGAZINE

www.dotnetcurry.com

SOFTWARE GARDENING: PRUNING

Add Notifications in your Website using jQuery

ECMA Script 6
New Objects and Updates to Existing Objects

Using **Task Runners Grunt and Gulp** in Visual Studio 2013 and 2015

Smart Unit Tests in Visual Studio 2015

Using Azure Storage API in an ASP.NET MVC Application

Visual Studio 2015:
Some Exciting New Features for Developers

FROM THE EDITOR

In this ever changing development world, the ability to build applications that target multiple platforms, is a must. Microsoft realizes this very well. From open sourcing .NET core and CLR; releasing previews of Visual Studio 2015 and .NET 4.6; hosting ASP.NET apps on Linux and Mac; supporting cross-platform development frameworks such as Xamarin, Apache Cordova and Unity; releasing a full featured Visual Studio Community Edition for Free; supporting native Android compilation; providing cross-platform tools for its ASP.NET web application framework and providing support for Grunt, Gulp, Bower, Bootstrap and many similar open source technologies, the message is clear; Microsoft wants to empower its developers with the best tools and technologies to write cross-platorm enterprise apps.

Visual Studio 2015 is at the heart of this edition's contents. Mahesh explores new features for developers in VS 2015, while Gouri talks about Smart Unit Testing. Ravi explores how to use front-end task runners like Grunt and Gulp in VS 2013 and 2015.

For our JavaScript audiences, Ravi talks about changes to the existing objects in EcmaScript 6 as well as new APIs in ES6, while I show how to build website notifications in jQuery with the ability to turn them off.

In our web-app development section, Mahesh shows how to use Windows Azure Storage API in an ASP.NET MVC application. To round off, Craig in his Software Gardening column talks about keeping software healthy by pruning it, what many know it by the name of Code Refactoring.

So how was this edition? E-mail me your views at suprotimagarwal@dotnetcurry.com

Suprotim Agarwal
Editor in Chief

Editor In Chief Suprotim Agarwal
suprotimagarwal@a2zknowledgevisuals.com

Art Director Minal Agarwal
minalagarwal@a2zknowledgevisuals.com

Contributing Writers Craig Berntson, Gouri Sohoni, Mahesh Sabnis, Ravi Kiran, Suprotim Agarwal

Reviewers Suprotim Agarwal

Next Edition 30th April 2015
www.dncmagazine.com

Visual Studio 2015:
Some Exciting New Features for Developers

In a world where apps target multiple platforms, Microsoft has been evolving its products and its framework in order to stay relevant. Its commitment to Open Source and enhancing developer experiences across platforms has been illustrated multiple times, the recent ones being the core CLR made open source, a full-featured free Community edition of Visual Studio 2013, the next version of open-source .NET that will also be released for Linux and Mac, Azure VMs on Linux and the next version of Visual Studio, called Visual Studio 2015 (currently in CTP 5 at the time of this writing)

Visual Studio 2015 showcases significant technology improvements that enhances the way developers work. Some improvements include Visual C++ for cross-platform development, the new open-source .NET compiler platform, C++ 11 and C++ 14 support, Visual Studio Tools for Apache Cordova, and ASP.NET 5, including regular feature updates.

As developers, we are constantly seeking ways to code efficiently and create apps and api's that are easy to build, consistent with standards, error free and maintainable. However there are a number of challenges that occur frequently in this process and hamper productivity. Some of these frequently encountered challenges are:

- Syntax Error handling
- Renaming methods, variables, etc.
- Debugging and Performance related issues.

Visual Studio provides solutions to most of these challenges and with every new release, new enhancements and features are released which makes the life of a developer easier. In the following article, we will discuss some new developer friendly features in Visual Studio 2015 which ultimately increases productivity.

This article uses an example that contains a Class Library Project and an ASP.NET Application. The ASP.NET Application refers to the class library to make database calls. The class library further makes use of Entity Framework for SQL Server Database Table Mapping.

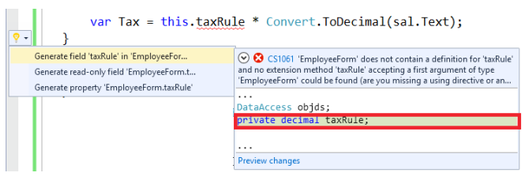
Let us get started.

Syntax Error Assistance for working with Missing Variable Errors

While working with .NET applications, it is a rule that we must declare a variable first, before using it. Consider the following code. We need to calculate tax for an Employee based upon some calculations:

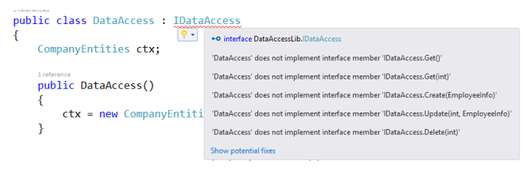
```
var Tax = this.taxRule * Convert.ToDecimal(sal.Text);
```

This code has a red squiggle indicating that *taxRule* is missing from the code. The **lightbulb icon** shown in the following image provides quick action for understanding the error and correcting it. This icon appears when you hover over the relevant code or when you have the cursor on the same line.

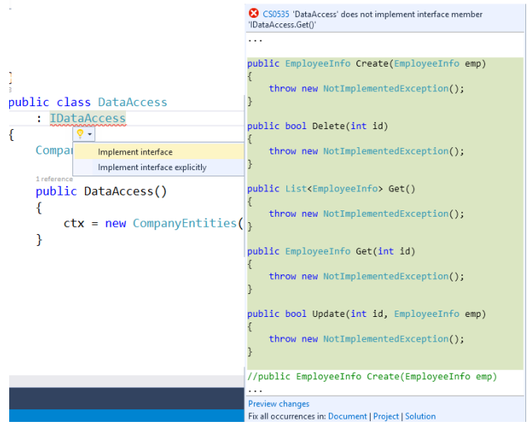


The Figure shows quick action preview for code refactoring. *taxRule* here can be generated as a field, read-only field or public property. Based on the selection, the *taxRule* will be declared in the code and the error will be removed.

Similarly we can make use of **Syntax Error Assistance** for a class implementing an interface.



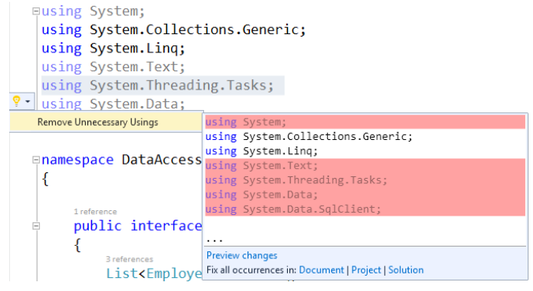
In the above code, the class *DataAccess* implements *IDataAccess* interface. VS 2015 IDE shows a lightbulb with quick action information which represents the methods from the interface that are *not* implemented by the class. On clicking on *Show potential fixes* at the bottom, VS 2015 offers to generate these methods for you:



The Syntax Error Assistance helps developers to reduce syntax errors that may occur in code, thereby improving productivity.

Using Code Suggestion

In VS 2015 IDE, the lightbulb icon provides code suggestions too. This provides help on possible code refactoring along with a preview for the same. E.g in a source code file, we may have several *unused* namespace references. As a good coding practice, we should eliminate the unnecessary namespaces as shown in the following figure.



The figure shows *gray* colored namespace references which are unused. The light bulb icons appears with the **Code Suggestions**. Drop down the icon (Ctrl + .) to see a preview of unused

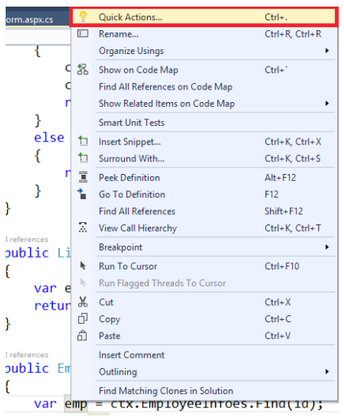
namespaces shown in pink. The advantage of the **preview changes** link is that developers get a clear idea of which namespaces are not necessary for the current code, and preview the changes before removing the unused namespaces. The figure also shows the **Fix all occurrences in Document |Project | Solution** option, which can be used to eliminate unnecessary references at current document, Project and Solution level.

Code Suggestion is also used to remove unnecessary code. Open the Dal.cs code file of the DataAccessLib project from the source code accompanying this article. In the DataAccess class, we have the following overloaded Get() methods:

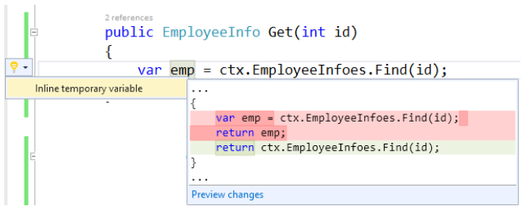
```
public List<EmployeeInfo> Get()
{
  var emps = ctx.EmployeeInfoes.
  ToList();
  return emps;
}

public EmployeeInfo Get(int id)
{
  var emp = ctx.EmployeeInfoes.Find(id);
  return emp;
}
```

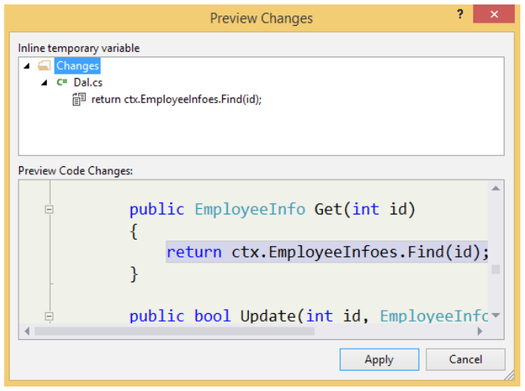
The above code declares inline temporary variables *emps* and *emp* to store resultant value. The methods return this temporary inline variable. We are not really using these variables for any other business operation, so as a good coding practice, we can eliminate these variables altogether using **Code Suggestions** Quick Actions. Right click on the *emp* declaration and select *Quick Actions* from the menu:



Clicking on 'Quick Actions' will bring up the Code Suggestions actions as shown here:



In the figure, we just saw that the Dark Pink highlights show possible code changes. On clicking on **Preview Changes**, the modified code will be displayed as shown here:



Clicking on 'Apply' modifies the code. We can repeat this for both Get() methods:

```
public List<EmployeeInfo> Get()
{
  return ctx.EmployeeInfoes.ToList();
}

public EmployeeInfo Get(int id)
{
  return ctx.EmployeeInfoes.Find(id);
}
```

As you observed, using VS 2015 IDE's Lightbulb **Syntax Error Assistance** and **Code Suggestions** help you to write error free and maintainable code.

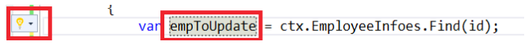
Renaming Experience

Our applications are divided into several layers (e.g. DataAccess, Business Logic, etc.) which contains classes, methods etc. in them. If we want to rename these classes and methods because of some reason, we need to check where these methods are used in the code, and rename them one-by-one. This is a time consuming step. In VS2015 IDE, Renaming has undergone an enhancement. We have been provided with **Renaming Experience**, which helps to easily rename types, methods, xml comments etc. in the source code.

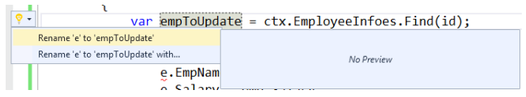
To explore this new enhancement, open Dal.cs in DataAccessLib project and locate the **Update()** method. The method has the following code:

```
public bool Update(int id, EmployeeInfo
emp)
{
  var e = ctx.EmployeeInfoes.Find(id);
  if (e != null)
  {
    e.EmpName = emp.EmpName;
    e.Salary = emp.Salary;
    e.DeptName = emp.DeptName;
    e.Designation = emp.Designation;
    ctx.SaveChanges();
    return true;
  }
  else
  {
    return false;
  }
}
```

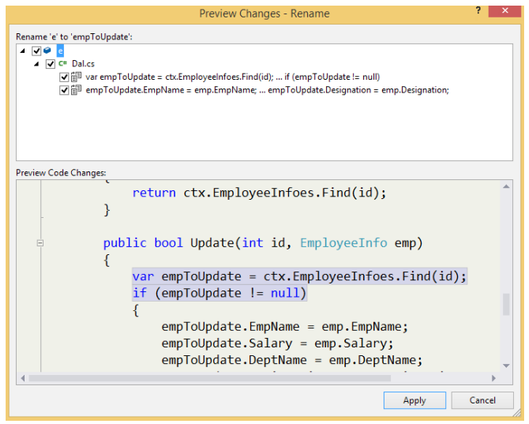
The variable 'e' is used to represent the searched employee based on the id. In order to provide a friendly name here for 'e' e.g. empToUpdate, rename 'e' to empToUpdate. Once the rename is done, the following code change will be displayed:



Visual Studio highlights all instances and the changed variable now has a gray rectangle. A quick action light bulb is displayed on the line. The Lightbulb displays the following options of renaming:



If we select the ***Rename 'e' to 'empToUpdate' with...*** the following preview gets displayed:



The preview shows us the expected modified code. The Quick Action allows us to immediately rename the variable.

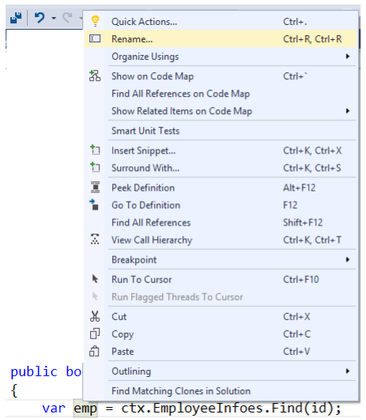
Renaming using Context Menu

We can also perform Renaming using the Context menu. To experience this, locate the Delete() method in the DataAccess class. This method has the following code:

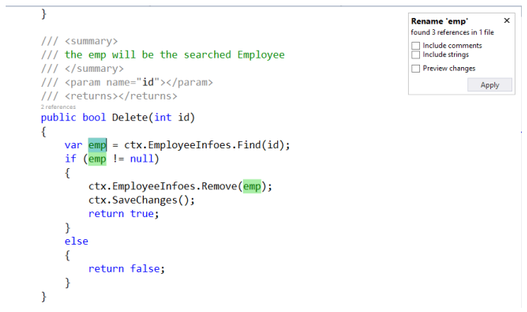
```
/// <summary>
  /// the emp will be the searched
  Employee
  /// </summary>
  /// <param name="id"></param>
  /// <returns></returns>
  public bool Delete(int id)
  {
    var emp = ctx.EmployeeInfoes.
    Find(id);
    if (emp != null)
    {
      ctx.EmployeeInfoes.Remove(emp);
      ctx.SaveChanges();
      return true;
    }
    else
    {
      return false;
    }
  }
```

The code uses 'emp' inline variable representing employee to delete. The same *emp* variable is used in comments decorated on the method. To rename 'emp' to 'empToDelete', right-click on the emp

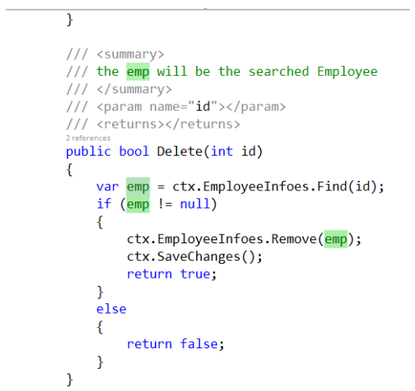
variable and select the 'Rename' option as shown in the following figure:



After Rename, the code will reflect the following effects:



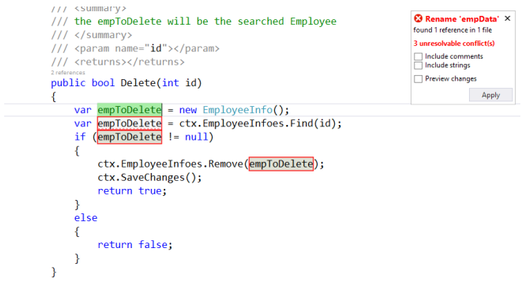
As you can observe, all 'emp' occurrences are highlighted and on the top-right, the Rename '*emp*' window is shown. This shows emp references with checkboxes for including Comments, strings, etc. If the Include comments is selected, the 'emp' mentioned in the comment on Delete method will also be selected as shown in following figure:



How cool is that! Here once we start renaming one instance of 'emp', the changes will be applied to all selected 'emp'. On clicking on *Apply* on the Rename 'emp' window that we just saw, the rename will be saved. But here we also need to think of renaming conflicts. What if in the same delete method, we have one more EmployeeInfo object declared? (See the following code)

```
public bool Delete(int id)
{
  var empData = new EmployeeInfo();
  var empToDelete = ctx.EmployeeInfoes.
  Find(id);
  if (empToDelete != null)
  {
    ctx.EmployeeInfoes.
    Remove(empToDelete);
    ctx.SaveChanges();
    return true;
  }
  else
  {
    return false;
  }
}
```

Here we have *empData*, a new object. If we try to rename it using **Renaming using Context menu** and if there is a name conflict, VS 2015 IDE will show errors:

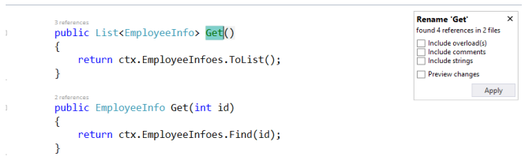


In such cases, just press **Escape** to get rid of renaming conflict issues.

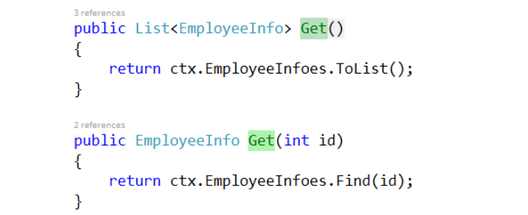
Renaming Methods

In the previous steps, we saw renaming inline variables. Now let's explore the renaming feature for methods. This is a major step because methods can usually be accessed across other layers of the application. So naturally renaming in one layer, should promote renaming in the calling layers too. In our Dal.cs (Data Access Layer), we have a Get() method. This is an overloaded method and it is accessed in the EmployeeForm.aspx.cs (Presentation layer) in an ASP.NET Application. Lets' see how the use of renaming feature in VS 2015 IDE helps rename method across layers.

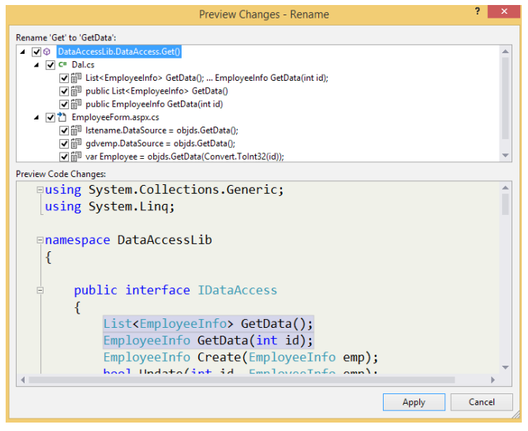
Open Dal.cs in DataAccessLib project and right click the Get() method, and select Rename. Alternatively we can also use Ctrl+R, Ctrl+R shortcut key combination for this. The following box appears:



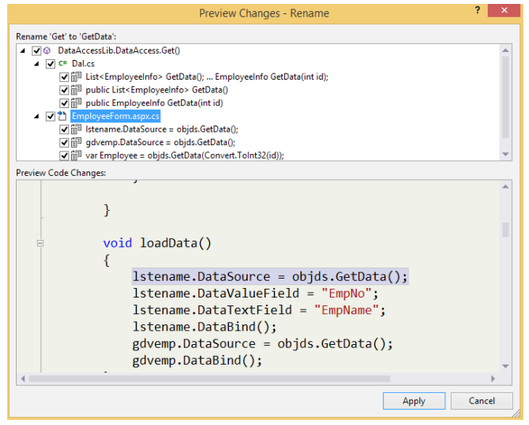
The overloaded Get() method is not selected. To select it from the box, select the *Include Overload(s)* checkbox. The overloaded Get() method will get selected as shown in the following figure:



Select the *Preview Changes* checkbox from the top-right dialog box and rename Get() to GetData() and click on *Apply* button. This will show the Preview Window as shown here:



Select EmployeeForm.aspx.cs in the top section of this window and preview code changes:



The VS 2015 IDE renaming helps to perform easy code management as far as renaming of variables and methods across layers are concerned.

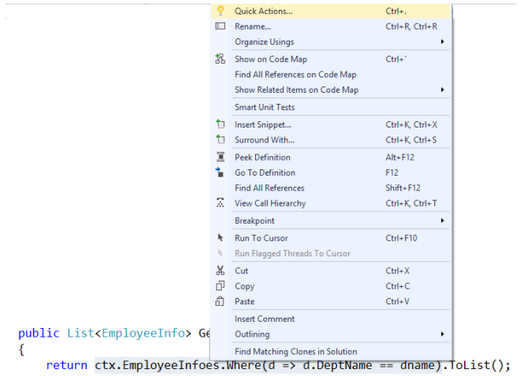
Code refactoring with inline local variables

In the **Code Suggestion** section, we have already seen how to eliminate unnecessary local variables. But consider a scenario where some methods have some logic written inside them:

```
public List<EmployeeInfo> GetData(string
dname)
{
  return ctx.EmployeeInfoes.Where(d =>
  d.DeptName == dname).ToList();
}
```

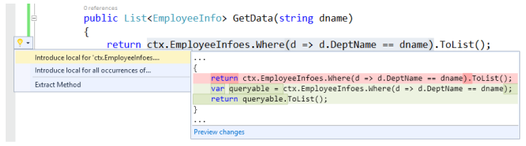
In the above code, the logic fires query on the EmployeeInfo and retrieves Employees based upon DeptName. The result is further converted into a list. (Feel free to consider an even more complex example here.).

Lets' make the logic simpler by separating the LINQ Query code and ToList() method. Select the above code, right click and select Quick Actions:



Note: *Code selected belongs to the LINQ query only.*

Clicking on Quick Actions will display the following options:



Select 'Introduce local for 'ctx.Employeeinfoes..." option. The code will be modified as shown here:

```
0 references
public List<EmployeeInfo> GetData(string dname)
{
    var queryable = ctx.EmployeeInfoes.Where(d => d.DeptName == dname);
    return queryable.ToList();
}
```

The code now becomes a little simpler to understand and is divided into two steps. The *queryable* variable name generated can be renamed as per the steps discussed in the *Renaming using context menu* section of this article.

Debugging Improvements in Visual Studio 2015 IDE

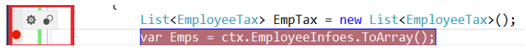
Configuration of the breakpoint in VS 2015 IDE has been improved and it has been made easier and provides a new user experience. An inline toolbar has been provided using which breakpoints can be enabled or disabled. Breakpoint configuration is provided with *conditions* and *actions*. *Conditions* tell the debugger to only pass the breakpoint when the debugger reaches to that line of code and when defined conditions are true. Actions define the operations to take place when the breakpoint condition is true.

Consider the following code in the Dal.cs file of DataAccessLib project, where each Employees Tax is calculated based on the salary and the resultant is stored in the List of EmployeeTax object.

```
public List<EmployeeTax> GetEmpTax()
{
  List<EmployeeTax> EmpTax = new
  List<EmployeeTax>();
  var Emps = ctx.EmployeeInfoes.
  ToArray();
  foreach (var item in Emps)
  {
    var emp = new EmployeeTax();
    emp.EmpNo = item.EmpNo;
    emp.EmpName = item.EmpName;
    emp.Salary = item.Salary;
    emp.Tax = item.Salary * Convert.
    ToDecimal(0.2);
    EmpTax.Add(emp);

  }
  return EmpTax;
}
```

Apply the breakpoint in the emps declaration statement (4th line) and you will observe that the breakpoint inline toolbar in the IDE gets displayed:



Breakpoint can be enabled/disabled using the small circles. Click on the Settings icon to bring up the breakpoint settings:

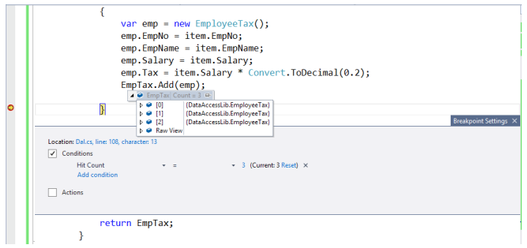
In the above code, now set a breakpoint at the closing curly brace of the foreach loop as shown in the following figure:

```
1 reference
public List<EmployeeTax> GetEmpTax()
{
    List<EmployeeTax> EmpTax = new List<EmployeeTax>();
    var Emps = ctx.EmployeeInfoes.ToArray();
    foreach (var item in Emps)
    {
        var emp = new EmployeeTax();
        emp.EmpNo = item.EmpNo;
        emp.EmpName = item.EmpName;
        emp.Salary = item.Salary;
        emp.Tax = item.Salary * Convert.ToDecimal(0.2);
        EmpTax.Add(emp);

    }
    return EmpTax;
}
```


Enter settings, pick window and define a condition that the breakpoint should apply when the *Htt Count* is 3, which means a breakpoint should get applied after the third iteration of the loop.

Run the application. Note, this method is called on the Button click event on the ASP.NET Web Form. You will observe that the breakpoint gets applied after the third iteration of the loop. This can be checked by using the *EmpTax* object.

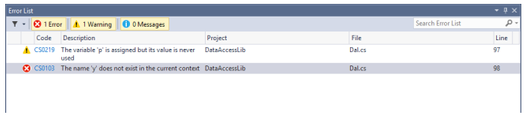


One nice feature in the VS 2015 IDE is that if we delete the breakpoint accidently, we can recover it back using Undo (Ctrl+z). The conditional application of breakpoint is useful for large recursive iterations. We can keep on adding conditions even when we are running the application. The Action part of the breakpoint settings allows us to specify the actions when the condition is true. Let's add an action that can log a message in the output window while debugging, by passing 'emp' to it:

And while we are at debugging, let me tell you that you can now debug Lambda expressions too, as we will see shortly!

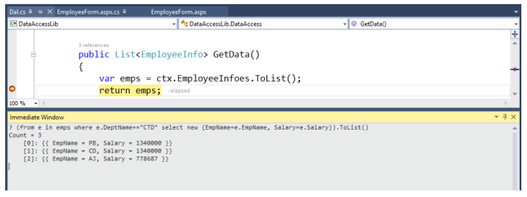
Error List Improvements

The Visual Studio 2015 IDE is improved with error lists. It shows compiler errors and code analysis warning in a single window. Compiler errors are prefixed by using **CS** and code analysis are prefixed using **CA**. The error window is displayed with compiler errors as shown here:



Improved Support for Lambda Expressions and LINQ query in the Immediate Window

When your code contains some large data stored in a collection, then it becomes difficult to debug and check values in that collection, if something were to go wrong. Visual Studio 2015 IDE solves this challenge by supporting Lambda Expressions and LINQ in the Immediate window. Yes you read that right! The Immediate window now allows to use Lambda Expressions and LINQ queries to filter data from collections. Lets' implement this. Apply breakpoint on the return statement of the GetData() method of the DataAccess class in Dal.cs. Run the application. When the breakpoint is hit, open immediate window and add LINQ queries in it as shown here.



And that's how we can filter data in the immediate window from the resultant collection using LINQ .

Using PerfView Tooltip

The Visual Studio 2015 IDE, provides a great performance tooltip feature, using which we can check how much time is taken by a statement. Typically we need this in case of foreach loops. We can see the PerfView tooltip while debugging as shown here:

```
1 reference
public List<EmployeeTax> GetEmpTax()
{

    List<EmployeeTax> EmpTax = new List<EmployeeTax>();
    var Emps = ctx.EmployeeInfoes.ToArray();
    foreach (var item in Emps) ≤4,142ms elapsed
    {
        var emp = new EmployeeTax();
        emp.EmpNo = item.EmpNo;
        emp.EmpName = item.EmpName;
        emp.Salary = item.Salary;
        emp.Tax = item.Salary * Convert.ToDecimal(0.2);
        EmpTax.Add(emp);

    }
    return EmpTax;
}
```

The above figure shows that the foreach loop takes nearly 4,142 ms.

Conclusion

This article explored some new IDE features in Visual Studio 2015 which makes developers more productive. Some of the features we explored were around suggestions for eliminating syntax errors, code refactoring, debugging, etc ■

Download the entire source code from our GitHub Repository at bit.ly/dncm17-vs2015newdevs

About the Author

Mahesh Sabnis is a Microsoft Certified Trainer (MCT) since 2005 and has conducted various Corporate Training programs for .NET Technologies (all versions). Follow him on twitter @maheshdotnet. Mahesh blogs regularly on .NET Server-side & other client-side Technologies at bit.ly/HsS2on

ECMA Script 6

New Objects and Updates to Existing Objects

This article is a continuation of the ES6 article published in the January edition of the DNC Magazine.

In the first part, we saw some improvements to the JavaScript language coming up in ES6. This article will focus on new object types as well as updated APIs of existing objects in the language.

As mentioned in the introduction of the first article, ES6 is designed to fit JavaScript better for writing larger applications. To do so, the language designers have added a number of new features that got inspired from typed substitutes of JavaScript and also from a number of other libraries, including some server-side libraries. Following are the new and updated objects at a glance:

- New data structures that make it easier to store unique values (Sets) or, key-value pairs with unique keys (Maps)
- Existing objects like Math and Number have got new capabilities to perform more operations as well as perform existing operations in a better way.
- *String* got some new functions to make parsing easier
- The type *Object* got functions to assign an object and compare two objects
- New functions on Array now makes it handy to find an entry, an index and to copy items inside the array
- New *Proxy* object to extend functionality of an existing object or function.

Platform Support for APIs

These APIs are not currently supported completely by all platforms. Latest versions of Chrome and Opera don't support some of the new functions on Strings and they don't support Proxy objects at all. Compilers like *traceur* and 6to5 also don't have polyfills for some of these APIs. I used Firefox nightly to test all the samples. You would still need to compile the scripts using *traceur*, as some of the scripts use the new syntax of ES6 for arrow functions and short hand functions.

Now that you got a brief idea on the API updates in ES6, let's start exploring each of them.

Set and WeakSet

Set

Set is a collection of distinct values of any JavaScript type (viz Number, Boolean, String, Object, etc.). It ignores any attempt made to insert a duplicate value. Sets are iterable; meaning we can loop over sets using *for...of* loop.

We can create a new instance of Set by calling constructor of Set type as follows:

```
var mySet = new Set(listOfItems);
```

..where *listOfItems* is an optional parameter, containing an iterable list of items to be inserted into the set. If not passed, an empty set will be created.

Following is an example of a set object:

```
var setOfObjects = new Set([17, 19, 38,
82, 17]);
```

We can loop over the *Set* object using a for...of loop, as *Set* is an iterable object. Following loop prints the values stored inside the Set:

```
for(let item of setOfObjects){
  console.log(item);
}
```

Check the output of this loop; it will not show duplicate entries added to the set. In our example, '17' is stored only once. Internally Set uses `SameValueZero(x,y)` to ignore duplicate entries. Now let's explore some methods provided by the Set API.

Adding Items

Items can be added to the Set using `set.add()` method.

```
setOfObjects.add(4);
setOfObjects.add(45);
setOfObjects.add(18);
setOfObjects.add(45);
```

The second attempt to insert 45 into the Set would be unsuccessful, as the Set already contains the value.

Verifying Presence of an Object

The `has()` method on the Set checks if the Set contains the object passed. The object is compared by its reference and not by value. Following example illustrates this:

```
var obj={value:100};
setOfObjects.add(obj);
console.log(setOfObjects.has(obj));
//true
console.log(setOfObjects.
has({prop:100}));  //false
```

Removing Objects

Objects stored in the Set can be deleted either by their reference using the `delete()` method or, by clearing all values using `clear()` method. Following are some examples:

```
setOfObjects.delete(obj);
setOfObjects.clear();
```

Size of Set

The `size` property on set holds the number of objects currently present in it.

```
console.log(setOfObjects.size);
```

Looping over Set

As mentioned earlier, a Set can be iterated using regular *for...of* loop. In addition to this, there are some other ways to iterate or, loop over the set. They are listed here: (* indicates that the methods return iterators)

- **entries()*: Returns an iterator containing key-value pair objects. Since they are same in case of Sets, each entry is an array with the corresponding value repeated in it

```
for(let item of setOfObjects.entries()){
  console.log(item);
}
```

- **values()*: Returns an iterator to iterate over values in the Set

```
for(let item of setOfObjects.values()){
  console.log(item);
}
```

- **keys()*:Returns an iterator to iterate over keys in the Set. As keys and values are same in Sets, the keys method produces same result as values does

```
for(let item of setOfObjects.keys()){
  console.log(item.);
}
```

- *forEach(callback)*: It is another way of looping over entries in the Set. Callback function is called for every entry in the Set.

```
setOfObjects.forEach(item => console.
log(item));
```

WeakSet

WeakSet is the weak counterpart of Set. A *WeakSet* doesn't prevent a value inserted into it from being garbage collected. The way it works is similar to Set, with the following exceptions:

- Can contain objects only. Numbers, Strings, Booleans, nulls and undefined values cannot be added into a WeakSet
- There is no way to iterate or loop over the values in a *WeakSet*, which means, methods like `values(), entries()` and `forEach()` are not available in *WeakSet*
- The set of operations one can perform over a *WeakSet* are: `add()`, `has()` and `delete()`. These methods work the same way as they do in *Set*

The reason behind the name *WeakSet* is, they don't prevent a value stored in them from being garbage collected.

Because of the restrictive nature of *WeakSet*, there are very few use cases where it can be used.

Map and WeakMap

Map

Maps are objects that are key-value pairs; both key and value can be any JavaScript object or value. Keys have to be unique in a given Map. Like *Sets*, *Maps* are iterable.

A new *Map* object can be created using constructor of the *Map* type as follows:

Following snippet shows how to create a Map with a set of objects:

```
var myDictionary = new Map([["key1",
"value1"], ["key2", "value2"]]);
```

As the dictionaries are iterable, we can loop over the items using *for...of* loop.

```
for(let dictionaryEntry of myDictionary)
{
  console.log(dictionaryEntry);
}
```

Map provides a set of methods to interact with it. Let's explore them.

Adding Items

New entries can be added to an existing *Map* using the `set()` method. This method checks if the key passed to it already exists in the *Map*. It adds the entry if the key is not found; otherwise discards the entry.

Following snippet adds some more entries to the *Map* created earlier:

```
myDictionary.set("key3", "value4");
myDictionary.set("key2", "value5");

var obj = {id: "1"};

myDictionary.set(obj, 1000);
myDictionary.set(obj, 1900);
```

An attempt to insert another entry with key assigned as *key2* as well as a second attempt to add an entry with *obj* as the key will be discarded as they already exist in *myDictionary*.

The keys are checked by reference, not by value. So, the following snippet will add an entry to *myDictionary*.

```
myDictionary.set({id: "1"}, 826);
```

Checking if a Key Exists

We can check if a key is already added to the *Map* using the `has()` method. Like in the case of *Sets*, the `has()` method checks for occurrence of the key by reference.

```
console.log(myDictionary.has("key2"));
//true
console.log(myDictionary.has(obj));
//true
console.log(myDictionary.has({id:
"1"}));   //false
```

Getting a Value by Key

It is possible to extract a value from a *Map* using `get()` method, if the key is known. If the key is not found in the *Map*, the method returns undefined.

```
console.log(myDictionary.get("key2"));
//value2
console.log(myDictionary.get("key2ii"));
//undefined
```

Removing Objects

The entries in a *Map* object can be either removed one by one using `delete()` method, or all entries can be removed at once using the `clear()` method. The `delete()` method takes key as the input and returns 'true' if the key is found and the entry is deleted; otherwise it returns 'false'.

Following are some examples of calling delete() and clear() methods:

```
console.log(myDictionary.delete({prop:
2000}));  //false
console.log(myDictionary.delete(obj));
//true
console.log(myDictionary.
delete("key1"));        //true

myDictionary.clear();
```

Size of Map

The `size` property on *Map* holds the number of objects currently present on it.

```
console.log(myDictionary.size);
```

Looping over Map

As mentioned earlier, *Maps* can be iterated using regular *for...of* loop. In addition to this, there are some other ways to iterate or, loop over the keys and values of the *Map*. They are listed below: (*indicates the methods return iterators)

- **entries()*: Returns an iterator containing key-value pair objects. Each entry is an array of length 2 with the first item being key and the second item being the value.

```
for(let item of myDictionary.entries()){
  console.log(item);
}
```

- **values()*: Returns an iterator to iterate over values in the Map

```
for(let item of myDictionary.values()){
  console.log(item);
}
```

- **keys()*:Returns an iterator to iterate over keys in the Map

```
for(let item of myDictionary.keys()){
  console.log(item);
}
```

- *forEach(callback)*: It is another way of looping over keys in the Map. Callback function is called for every key.

```
myDictionary.forEach(item => console.
log(item));
```

WeakMap

WeakMap works similar to the way *Map* works, with a few exceptions. The exceptions are identical to the exceptions in case of *WeakSet*. *WeakMaps* don't restrict the objects used as keys from being garbage collected. Following are the list of characteristics of a *WeakMap*:

- Keys can only be objects; they cannot be of value types. Values can be of any type

- They don't support iteration over the items. So, *for...of* loop cannot be used over the items in a *WeakMap* and the methods `entries(), values()` and `keys()` are not supported

- The set of operations supported are: `set(), get(), has()` and `delete()`. Behavior of these operations is same as their behavior in case of *Maps*.

Numbers

Some of the global number functions like `parseInt(), parseFloat()` have been moved to the *Number* object and now the language has also got better support for dealing with different number systems. Let's see these changes in action.

Number Systems

ES6 defines an explicit way to work with Octal and Binary number systems. Now it is easier to represent these numbers and also to convert the values between these number systems and Decimal number system.

All octal numbers have to be prefixed with "0o". It is also possible to convert a string following this format, to a number using the *Number* object. Following are some examples:

```
var octal = 0o16;
console.log(octal); //output: 14

var octalFromString = Number("0o20");
console.log(octalFromString); //output:
16
```

Similarly, any binary number has to be prefixed with "0b". They can also be converted from string using the *Number* object.

```
var binary = 0b1100;
console.log(binary);    //output: 12

var binaryFromString =
Number("0b11010");
console.log(binaryFromString);
//output: 26
```

parseInt and parseFloat

Now, the `parseInt()` and `parseFloat()` functions are made available through *Number* object. Using these functions through the *Number* object is more explicit. They work the same way as they used to earlier.

```
console.log(Number.parseInt("182"));
console.log(Number.
parseFloat("817.12"));
```

isNaN

Now, we can check if an expression is a valid number using `isNaN()` method of Number. The difference between the global `isNaN()` function and the `Number.isNaN` is, the former converts value to number before checking if it is a number. Following are some examples:

```
console.log(Number.isNaN("10"));
//false as "10" is converted to the
number 10 which is not NaN

console.log(Number.isNaN(10));
//false
```

"NaN" means "this value is a numeric Not-a-Number value according to IEEE-754"

Fyi, another way to determine if the value is of the number type is by using `typeof()`.

isFinite

This method finds out if a value is a finite number. It tries to convert the value to number before checking if it is finite. Following are some examples of using this method:

```
console.log(Number.isFinite("10"));
//false
console.log(Number.isFinite("x19"));
//false
```

isInteger

This method checks out if a value is a valid integer. It doesn't convert the value to number before checking if it is an integer. Following are some examples of using this method:

```
console.log(Number.isInteger("10"));
//false
console.log(Number.isInteger(19));
//true
```

Constants

The *Number* API now includes 2 constant values:

- *EPSILON* (Smallest possible fractional number). Its value is 2.220446049250313e-16

- *MAX_INTEGER* (Largest possible number). Its value is 1.7976931348623157e+308

Math

The new methods added to Math object finds logarithmic values, contains hyperbolic trigonometric functions and a few other utility functions. Following is the list of methods added to Math.

Logarithmic Functions

- *log10*: Calculates logarithm of the value passed with base 10

- *log2*: Calculates logarithm of the value passed with base 2

- *log1p*: Calculates natural logarithms of the value passed in after incrementing the number by 1

- *expm1*: It does the reverse of the previous function. Raises the value of input with exponent of natural logarithm and subtracts the result by 1

Hyperbolic Trigonometric Functions

- *sinh, cosh, tanh*: Hyperbolic sine, cosine and tangent functions respectively

- *asinh, acosh, atanh*: Inverse hyperbolic sine, cosine and tangent functions respectively

Miscellaneous Functions

- *hypot*: Accepts two numbers and finds value of hypotenuse of a right angle triangle of which the values passed are adjacent sides of right angle

- *trunc*: Truncates fractional part of the value passed in

- *sign*: Returns sign of the value passed in. if NaN is passed, results NaN, -0 for -0, +0 for +0, -1 for any negative number and +1 for any positive number

- *cbrt*: Finds cubic root of the value

String

String Templating

In every JavaScript application, we use strings heavily and in many cases, we have to deal with appending values of variables with strings. Till now, we used to append variables using the plus (+) operator. At times, it is quite frustrating to deal with such cases. ES6 brings support for String Templating to address this difficulty.

If we use templates, we won't have to deal with breaking the strings and attaching variables to them. We can continue writing the strings without breaking the quotes. To use this feature, we cannot use single or double quotes; we need to use back quotes (`). Following example shows the syntax of using templates. It appends value of a variable to a string to form path of a REST API:

```
var employeeId = 'E1001';
var getDepartmentApiPath = `/api/
department/${employeeId}`;

console.log(getDepartmentApiPath);
```

You can use any number of variables in templates. Following example forms another API path using two variables:

```
var projectId = 'P2001';
var employeeProjectDetailsApiPath = `/
api/project/${projectId}/${employeeId}`;

console.
log(employeeProjectDetailsApiPath);
```

We can perform some simple arithmetic operations inside the templates. Following snippet shows them:

```
var x=20, y=10;

console.log(`${x} + ${y} = ${x+y}`);
console.log(`${x} - ${y} = ${x-y}`);
console.log(`${x} * ${y} = ${x*y}`);
console.log(`${x} / ${y} = ${x/y}`);
```

Utility Functions

String adds one utility function `repeat()` in ES6. This function repeats the string for a specified number of times and returns it. It can be called using any string.

```
var thisIsCool = "Cool! ";
var repeatedString = thisIsCool.
repeat(4);
console.log(repeatedString);
```

Substring Matching Functions

ES6 adds the functions `startsWith()`, `endsWith()` and `includes()` to String's prototype to check for occurrence of a substring inside a given string at the beginning, towards end or at any position respectively. All these functions return Boolean values. The function `includes()` can be used to check for occurrence at a given index in the string as well. Following are examples showing usage of these functions:

- *startsWith()*:

```
console.log(repeatedString.
startsWith("Cool! "));

console.log(repeatedString.
startsWith("cool! "));
```

- *endsWith()*:

```
console.log(repeatedString.
endsWith("Cool! "));
console.log(repeatedString.
endsWith("Cool!"));
```

- *includes()*:

```
console.log(repeatedString.
includes("Cool! "));
console.log(repeatedString.
includes("Cool! ", 6));
console.log(repeatedString.
includes("Cool! ", 10));
```

Unicode Functions

ES6 adds functions to find Unicode equivalent of characters, to convert characters to Unicode and to normalize a Unicode string using different compositions.

- *codePointAt()*: Returns Unicode character of character at the specified position in the string

```
console.log(repeatedString.
codePointAt(0));
```

- *fromCodePoint()*: It is a static function on string. It returns character equivalent of the Unicode passed as an argument to it.

```
console.log(String.fromCodePoint(200));
```

- *normalize()*: Returns Unicode normalization form of the string. It accepts the format to be used for normalization. If format is not passed, it uses NFC. Check documentation on MDN for more details on this function.

```
"c\u067e".normalize("NFKC");  //"cie"
```

Array

Arrays are the most commonly used data structures in any language. ES6 brings some new utility functions to objects of *Array* type and also adds some static methods to *Array* to make searching elements, copying elements in the same *Array*, iterating over the elements and converting non-*Array* types to *Array* types.

Iterating Over Array

Like in case of *Maps*, *Arrays* now have the methods `entries()` and `keys()` to iterate over the values.

- **entries()*: Each entry returned from the entries function is an array of two elements containing a key and its corresponding value. For Arrays, keys are same as the indices.

```
var citiesList = ["Delhi", "Mumbai",
"Kolkata", "Chennai", "Hyderabad",
"Bangalore"];

for(let entry of citiesList.entries()){
  console.log(entry);
}
```

- **keys()*: Keys are same as indices; so this function returns index of every item in the array.

```
for(let key of citiesList.keys()){
  console.log(key);
}
```

Finding Occurrence

Arrays now have two methods, *find* and *findIndex*, that take a predicate and return the item in the *Array* that matches the condition checked by the predicate. For the predicate, we can pass an arrow function.

- *find()*: Accepts a predicate and returns the first item in the array that satisfies the condition checked by the predicate.

```
console.log(citiesList.find( city =>
city.startsWith("M") ));
```

- *findIndex()*:Accepts a predicate and returns index of the first item in the array that satisfies the condition checked by the predicate.

```
console.log(citiesList.findIndex( city =>
city.startsWith("M") ));
```

Filling and Copying

It is now easy to fill the entire *Array* or, a part of the *Array* with an item and also copy a portion of the *Array* into rest of it.

- *fill()*: Following is the syntax of the fill() function:

```
arrayObject.fill(objectToFill,
startIndex, endIndex);
```

Only first argument is mandatory. When it is called with just one argument, it fills the entire array with the value passed in.

```
citiesList.fill("Pune");

citiesList.fill("Hyderabad", 2);

citiesList.fill("Bangalore", 3, 5);
```

- *copyWithin()*: Copies one or more elements inside the array to other positions in the array.

```
citiesList.copyWithin(0, 3);
//elements at 0 to 2 into elements from
3 onwards

citiesList.copyWithin(0, 3, 5);
//elements at 0 to 2 into elements from
3 to 5

citiesList.copyWithin(0, -3);
//negative index starts from end of the
array
```

Converting to Array

ES6 adds two static methods to *Array* that convert collections and stream of data into *Arrays*.

- *of()*: This function takes a list of objects and returns an Array with these objects as items in it.

```
var citiesInUS= Array.of("New York",
"Chicago", "Los Angeles", "Seattle");
```

- *From()*: Used to convert Array-like data (viz., arguments of functions) into arrays.

```
function convertToArray(){
  return Array.from(arguments);
}
var numbers = convertToArray(19, 72, 18,
71, 37, 91);
```

Object

Object got two new static functions in ES6 - to compare two objects and assign enumerable properties of a number of objects into one object.

- *is()*: Accepts two objects and returns a Boolean value representing if the objects are equal

```
var obj = {employeeId: 100};
var obj2 = obj;

console.log(Object.is(obj, {employeeId:
100}));     //false
console.log(Object.is(obj, obj2));
//true
```

- assign(): Following is the syntax of this function:

```
Object.assign(target, source1, source2,
...)
```

Assigns enumerable properties of all source objects into the target object.

```
var obj3 = {departmentName: "Accounts"};

var obj4 = {};
Object.assign(obj4, obj, obj3);
//contents of obj4: {employeeId: 100,
departmentName: "Accounts"}
```

Proxy

As name of the object itself suggests, the *Proxy* object is used to create proxies around objects and methods. The *Proxy* object is very useful to perform tasks like validations before calling a function and format data of a property when its value is accessed. In my opinion, *Proxies* define a new way to decorate the objects in JavaScript. Let's see it in action.

Consider the following object:

```
var employee={
  employeeId: 'E10101',
  name:"Hari",
  city:"Hyderabad",
  age: 28,
  salary: 10000,
  calculateBonus(){
    return this.salary * 0.1;
  }
};
```

Proxying Getters

Let's format the value of this employees salary when it is accessed. For this, we need to define a proxy around getter of the object properties and format the data. Let's define a Proxy object to do this task:

```
var employeeProxy = new Proxy(employee,
{
  get(target, property){
    if(property === "salary"){
      return `$ ${target[property]}`;
    }
    return target[property];
  }

});

console.log(employeeProxy.salary);
```

As you can see, the `get()` method of the proxy object takes two arguments:

- target: Object on which the getter is being redefined
- property: Name of the property to be accessed

Take another look at the snippet. I am using two ES6 features to compose it: short-hand way of defining function and string templates.

Proxying Setters

EmployeeId of an employee should be assigned only once. Any further attempt to assign a value to it should be prevented. We can do it by creating a proxy around setter on the object. Following snippet does this:

```
var employeeProxy = new Proxy(employee,
{
  set(target, property, value){
    if(property === "employeeId"){
      console.error("employeeId cannot
      be modified");
    }
    else{
      target[property] = value;
    }
  }
});

employeeProxy.employeeId = "E0102";
//Logs an error in the console
```

Proxying Function Calls

Let us assume bonus of the employee has to be calculated only if salary of the employee is greater than $16,000. But the `calculateBonus()` method in the above object doesn't check this condition. Let's define a proxy to check this condition.

```
employee.calculateBonus = new
Proxy(employee.calculateBonus, {
  apply(target, context, args){
    if(context.salary < 15000){
      return 0;
    }
    return target.apply(context, args);
  }
});

console.log(employee.calculateBonus());
//Output: 0

employee.salary=16000;
console.log(employee.calculateBonus());
//Output: 1600
```

Conclusion

As we saw, ES6 introduces a number of new APIs on existing objects and also a set of new objects and data structures that ease a lot of work. As already mentioned, some of these APIs are not supported by all platforms, as at the time of writing this article. Let's hope to see them supported in near future. In future articles, we will explore promises and modules in ES6 ■

Download the entire source code from our GitHub Repository at bit.ly/dncm17-es6newapis

About the Author

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

SOFTWARE GARDENING | Pruning

Code Refactoring

Imagine that you have a beautiful rose garden. It takes many hours of work to keep it beautiful and the roses healthy. A key aspect of growing roses is *pruning*. That is, you need to cut out the dead and dying flowers so that the young, healthy buds can thrive. Pruning roses also gives them a more attractive shape.

But pruning can also be applied elsewhere. For example, in the timber industry, where entire mountainsides of forest need to be kept healthy, different types of pruning are used. The first of these is selective cutting, where specific types of trees are identified and only those are cut. Then there is thinning, where large trees are cut out of thick heavy forests, so the small trees that are struggling can grow and be healthy. Finally, there is clear-cutting, which is used to completely clear a side of a mountain of all trees. Certain types of trees will not survive if selective cutting or thinning is used, so clear-cutting is the only alternative. Also, if a forest is attacked by disease or insects, clear-cutting may be used to remove a section of forest in an effort to save the entire forest. In all cases, the timber company replants trees so that new growth will keep the forest healthy. In all cases, the forest is still a forest. Pruning has not changed the functionality of what nature gave.

Do you prune your software? Just like the roses or the forest, pruning is necessary to keep software healthy. As software gardeners, our term for pruning is *refactoring*. The first important thing you need to know about refactoring is that it isn't rewriting. Refactoring is defined as:

> "the process of restructuring existing computer code – changing the factoring – without changing its external behavior"

Src: http://en.wikipedia.org/wiki/Code_refactoring

Let's think back to a past column I wrote about unit testing. The process of unit testing is shown in Figure 1. You can see that refactoring is a key part of unit testing. And, not surprisingly, unit testing is a key part of refactoring. After all, how do you know if you haven't changed the external behavior of the piece of code, without testing?

Code Smells

The most famous book on refactoring was written by Martin Fowler. Its title is "Refactoring: Improving the Design of Existing Code". The book is a catalog of refactorings and includes a chapter called "Bad Smells in Code" that discusses ways to identify and fix problematic code that could be a good candidate for refactoring. The smells include duplicated code, long methods, large classes, and long parameter list. (I once had to update code written by someone else and found a class with a constructor that had 144! parameters). There are nearly 20 more code smells listed by Martin.

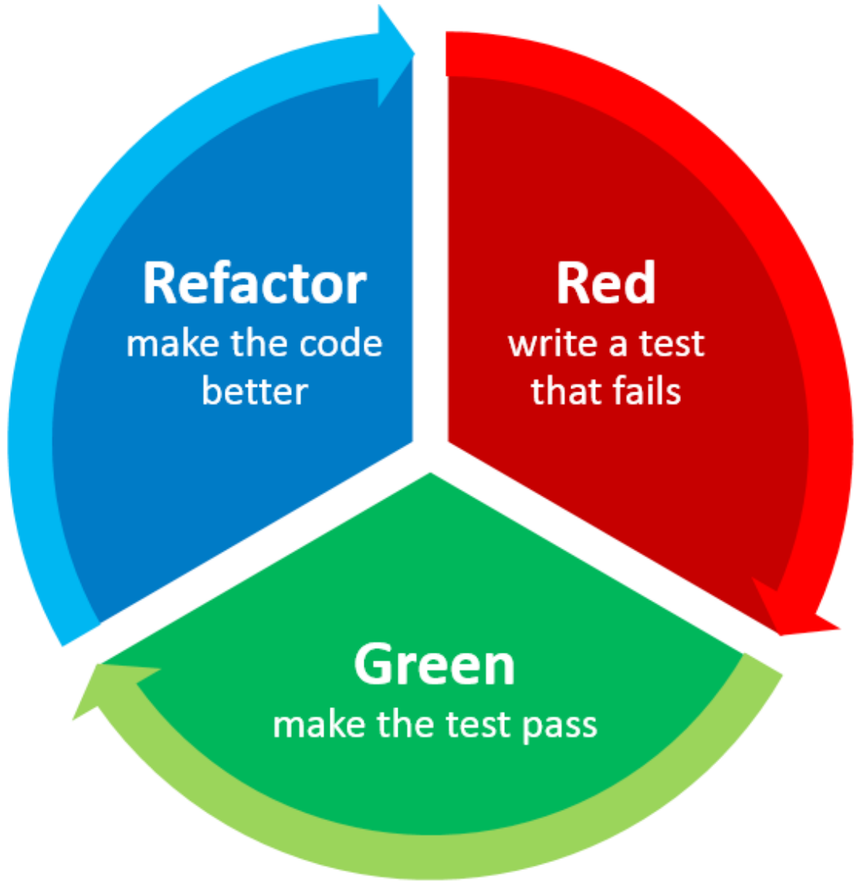


Figure 1: The unit testing process includes refactoring.

In addition to describing code smells and cataloging refactoring patterns, the book gives reasons to refactor: improve the design of software, make software easier to understand, help you find bugs, and help you to program faster. This last one seems contradictory. How can you program faster if you have to refactor the code? The answer is that refactoring makes it easier to read the existing code.

Martin Fowler also lists what he calls, "The Rule of Three", which are three rules for when to refactor. First, you should refactor when you add function. Second, refactor when you need to fix a bug. And third, refactor as you do a code review.

Once you've identified the specific code smell, the book points you to refactorings that can be used to fix it. You can fix the code with copy and paste. But there is a better way. Visual Studio has some built-in refactoring tools that are easier to use than copy and paste.

Hover over a variable in the Visual Studio editor (for this column, I used the free Visual Studio 2013 Community Edition, which is functionally equivalent to VS Professional), then right-click and select Refactor. You'll see that six refactorings are available. You will also notice that each refactoring has a keyboard shortcut that begins with Ctrl+R.

Figure 2: The Visual Studio refactoring menu

In the rest of this column, I will take you through each of these refactorings.

Refactoring: Rename

The late computer scientist Phil Karlton said, "There are only two hard things in Computer Science: cache invalidation and naming things." (A variation says "There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors.") How often have you named a class, variable, or other object and after sometime realized that the name you chose turns out to be wrong? How often have you needed to name something and wasted time trying to come up with just the right name for it before even typing it into your editor? Often times the correct name comes to you only after you've seen how it will be used in the code. This is where Rename Refactoring comes into play.

Martin Fowler describes Rename Method as,

> "An important part of the code style I am advocating is small methods to factor complex processes. Done badly, this can lead you on a merry dance to find out what all the little methods do. The key to avoiding this merry dance is naming the methods...Remember you code for a human first and a computer second

(Refactoring, p. 273)

In Visual Studio, the Rename Refactoring works on more than just a method. It can be used on variables, classes, even namespaces. Place the cursor on the object to be renamed and press Ctrl+R,Ctrl+R (think of this as Refactor, Rename). The Rename dialog appears.

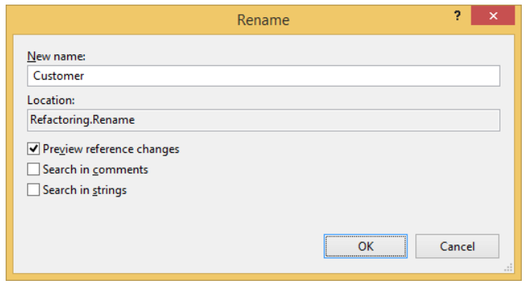


Figure 3: The rename dialog is used for the rename refactoring.

Enter the new name and select the options you want, then select OK. If you checked Preview reference changes, you will get an additional dialog showing you where each refactoring takes place and what the changes will look like. You can then unselect the places you don't want the refactoring to occur.

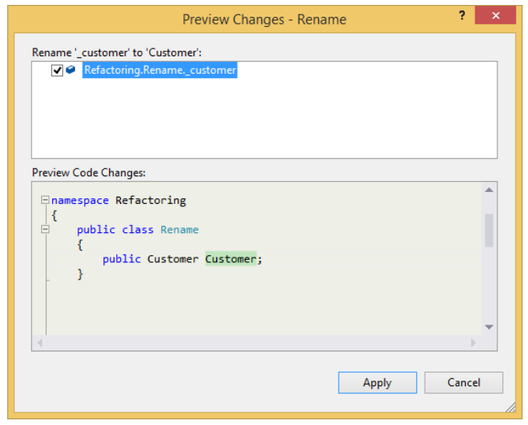


Figure 4: The Preview Changes dialog

Click Apply in the Preview Changes dialog and the refactorings will be made.

Refactoring: Extract Method

One of the rules for good methods is that a method should do only one thing. Also, a good method will be short and not require you to scroll though lots of code. This next refactoring, Extract Method helps you fix these issues by removing code and putting it in its own method. You can also use this refactoring to simplify code.
From Martin Fowler:

> "Extract Method is one of the most common refactorings I do. I look at a method that is too long or look at code that needs a comment to understand its purpose. I then turn that fragment of code into its own method.

(Refactoring, p. 110)

To use this refactoring, highlight the code to extract, then press Ctrl+R,Ctrl+M (Refactor, Method). Here's some real code from one of my projects.

```
if (viewModel.NewEmailParentTable.
ToLower() == "company" && viewModel.
NewEmailIsDefault)
{
  var companyEmailAddresses =
  _context.CompanyEmailAddresses.
  Include("EmailAddress");

  if (companyEmailAddresses.Any())
  {
    var currentDefault =
     companyEmailAddresses.
     FirstOrDefault(e => e.EmailAddress.
     IsDefault).EmailAddress;
    if (currentDefault.Id != viewModel.
    NewEmailAddressId)
    {
      currentDefault.IsDefault = false;
      _context.EmailAddresses.
      Attach(currentDefault);

     _context.Entry(currentDefault).
      State = EntityState.Modified;
```

```
      _context.SaveChanges();
    }
  }
}
```

After I wrote the code and it passed unit tests, I went back to refactor. The original method was about three screens long. This piece of code has a smell in that it has several nested *if* statements. I highlighted the entire code section and pressed Ctrl+R,Ctrl+M. The Extract Method dialog was displayed.

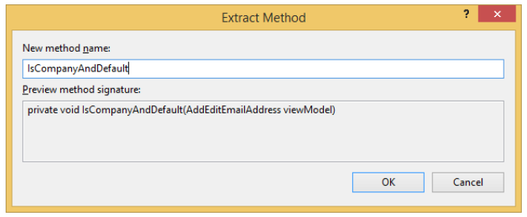


Figure 5: The Extract Method dialog.

Notice how the refactoring tools figured out what parameters the method needed and set those up automatically. I entered the name for the new method and clicked OK. Visual Studio did all the work of creating the method in my class and replacing the selected code with a single line of code. I then refactored the nested *if* statements. Later on, I decided my original name wasn't quite right and used the Rename refactoring to change the name of the method. In the end, the code is easier to read and maintain.

Refactoring: Encapsulate Field

The next refactoring, *Encapsulate Field*, turns a field into a property. Look at this code

```
public class Line
{
  public int length, width;
}

public class MyLine
{
  public static void Main()
  {
    Line line = new Line();
    line.length = 12;
    line.width = 1;

    // Some code to draw the line on the
    screen
  }
}
```

Length and width are technically fields, not properties because they don't have getters or setters. Let's change that, but first, here's what Martin Fowler has to say,

> "One of the principal tenets of object oriented programming is encapsulation, or data hiding. This says that you should never make your data public. When you make data public, other objects can change and access data values without the owning object's knowing about it. This separates data from behavior... Encapsulate field begins the process by hiding the data and adding accessors. But this is only a first step. Once I've done Encapsulate Field, I look for methods that use the new methods to see whether they fancy packing their bags and moving to the new object with a quick Move Method.
>
> (Refactoring, p. 206)

Now let's look at how to do this refactoring in Visual Studio. Put the cursor on length in the Line class then press Ctrl+R,Ctrl+E (Refactor, Extract). The Encapsulate Field dialog is displayed.

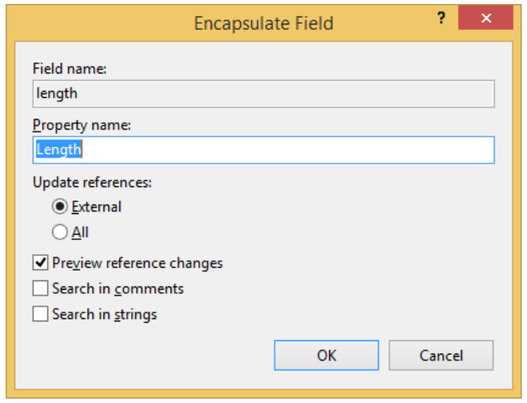


Figure 6: Use the Encapsulate Field dialog to change a field to a property.

Visual Studio recommended 'Length' as the property name. You can change it to something else if you want. When you click OK, a Preview dialog is displayed. Click Apply there. Here's the refactored code.

```
public class Line
{
  public int length, width;

  public int Length
  {
    get { return length; }
    set { length = value; }
  }
}

public class MyLine
{
  public static void Main()
  {
    Line line = new Line();
    line.Length = 12;
    line.width = 1;

    // Some code to draw the line on the
    screen
  }
}
```

This is nice and tidy and follows a common standard that property name begin with a capital letter.

Refactoring: Extract Interface

Interfaces are a terrific way to get reuse and inheritance in your code. They also make code easier to unit test because it's quite easy to mock an interface. The Extract Interface refactoring is very useful for that legacy code that you need to test and it looks at the class and creates an interface for it.

Here's Martin Fowler's take: "
"Classes use each other in several ways. Use of a class often means ranging over the whole area of responsibilities of a class. Another case is use of only a particular subset of a class's responsibilities by a group of clients. Another is that a class needs to work with any class that can handle certain requests...Interfaces are good to use whenever a class has distinct roles in different situations. Use Extract Interface for each role." (Refactoring, pp 341-342)

Let's see this refactoring in action. Here's the original method. Note that I've left out the actual implementation code.

```
public class CustomerRepository
{
  public Customer Find(int id) {}
  public IQueryable<Customer> GetAll() {
}
  public void InsertOrUpdate(Customer
  customer) { }
  public void Save() { }
  public void Delete(int id) { }
  public Customer
  GetByPartialName(string name) { }
}
```

Place the cursor on the class name and press Ctrl+R,Ctrl+E (Refactor, Extract) to display the Extract Interface dialog. Visual Studio suggests a name for the Interface and allows you to select which methods to include. I selected all except GetByPartialName. When you click OK, Visual Studio created a new file (ICustomerRepository) and placed it in the same folder as the original class.

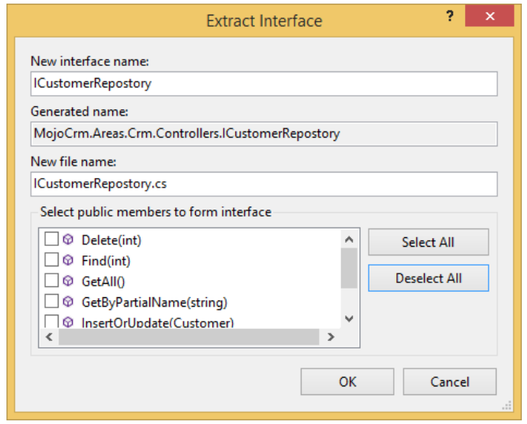


Figure 7: Extract Interface Dialog

```
interface ICustomerRepostory
{
  void Delete(int id);
  Customer Find(int id);
  IQueryable<Customer> GetAll();
  void InsertOrUpdate(Customer
  customer);
  void Save();
}
```

The original class is then changed to inherit from the new interface.

```
public class CustomerRepostory :
ICustomerRepostory
{
  public Customer Find(int id) {}
  public IQueryable<Customer> GetAll() {
}
  public void InsertOrUpdate(Customer
  customer) { }
  public void Save() { }
  public void Delete(int id) { }
  public Customer
  GetByPartialName(string name) { }
}
```

Now it's easier to mock the code or substitute a different repository implementation for this one.

Refactoring: Remove Parameters

Getting all the parameters you need for a particular method is hard. As you develop the functionality, you realize that some parameters aren't needed at all. Others may turn into properties or fields. It's easy to do this if the method is called one or two times, but what if you call it many times from many places in the project? Then it becomes more difficult. This next refactoring helps you remove parameters.

Sticking with the pattern of this discussion, here's Martin Fowler's explanation: *"A parameter indicates information that is needed; different values make a difference. Your caller has to worry about what values to pass. By not removing the parameter you are making further work for everyone who uses the method."* (Refactoring, p. 277)

To use this refactoring, place the cursor over either the method definition or usage and press Ctrl+R, Ctrl+V (Refactor, remoVe) to display the Remove Parameters dialog.

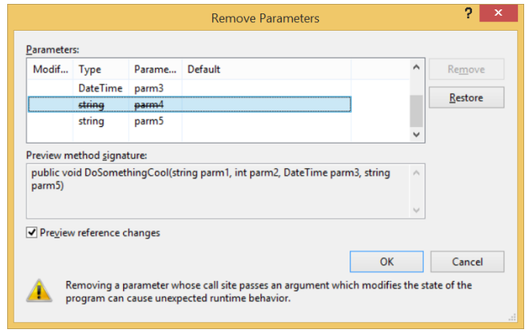


Figure 8: It's easy to remove parameters with the Remove Parameters dialog.

Select the parameter to remove and press Delete. The parameter you want to remove is then shown in strikeout font and a preview of the method is shown. Click OK. If you selected Preview reference changes, a second dialog showing the changes is displayed. Click Apply. Visual Studio then refactors out the parameter.

Refactoring: Reorder Parameters

Do you think about the order of parameters in a method? Does it really matter? Good coding practices says it does. You should order parameters from most important to least important and optional parameters should fall at the end. Now, you may have a differing opinion and that's okay.

But this refactoring helps you get parameters in the order needed. Interestingly, this is one refactoring pattern Martin Fowler doesn't catalog.
Now on to the usage. Place the cursor on the method definition or usage and press Ctrl+R, Ctrl+O (Refactor, reOrder). The Reorder Parameters dialog is displayed.

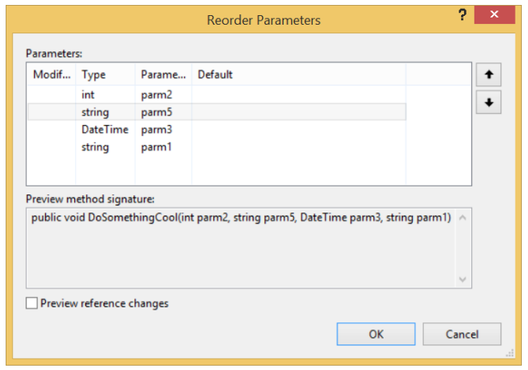


Figure 9: Changing parameter order with the Reorder Parameters dialog.

Select the parameter to move, then use the mover arrows on the right. You can optionally preview changes. Click OK, then Apply if you previewed changes. Again, Visual Studio does all the updates for you.

Summary

There you have it. You've learned a bit about what refactorings are and why you want to refactor your code. I have shown you the built-in refactorings in Visual Studio.

If you look at the Martin Fowler book, you'll see dozens of refactorings. He also has an online catalog with even more refactorings at http://refactoring.com/catalog/.

If you want more automatic refactorings, look to tools like Resharper or CodeRush. These commercial tools add enhancements to the built-in Visual Studio refactorings and add even more refactorings.

Refactoring your code is important to keep it maintainable and easy to read. It also helps reduce potential bugs and could improve your application's performance. It's pretty clear that refactoring is a practice that will keep your code 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.

Smart Unit Tests in Visual Studio 2015

Over the years, we have seen a lot of enhancements in different versions of Visual Studio. Visual Studio provides many tools like the Static Code Analysis, Code Metrics, IntelliTrace, Code Profiler, Unit Testing mechanism, Code Map etc. that facilitates writing code the right way. All these tools come integrated in Visual Studio, depending on the version you are using. You can read an article authored by Subodh which talks about Code Optimization Tools in Visual Studio 2013.

Microsoft is now on the verge of releasing a newer version of Visual Studio i.e. Visual Studio 2015 which boasts of features which will make it even more simpler to create high quality code without any compromise on productivity. In this article, I will introduce you to one such feature called Smart Unit Tests in Visual Studio 2015.

Unit Testing and Visual Studio

Unit Testing helps in ensuring that the code is working as expected. Testing helps in finding out the quality of the software. The earlier you start testing your code, the easier it becomes to evaluate the quality of code. Unit Testing feature was available from Visual Studio 2005 as an integral part of MSTests. Over the years a lot of enhancements have been made to it - like auto generation of Unit Tests in Visual Studio 2010, PEX framework as a separate installable for generating parameters since Visual Studio 2010, support for third party unit tests framework from Visual Studio 2012 and so on. With Visual Studio 2015, unit testing goes to the next level with the introduction of Smart Unit Tests.

Smart Unit Tests

Developers need to write unit tests for a variety of reasons. Some of these reasons are as follows:

- To maintain high quality code. Teams would like to release the software with fewer bugs. Writing unit tests reduces the bugs in new features as well as existing features

- Teams need to deliver code early. Contrary to the belief, writing unit tests makes development faster

- Writing unit tests for code makes your code more testable for non-functional testing like performance testing

- Software which is not sufficiently tested becomes more bugs prone, which leads to more time in creating error free code, thereby increasing costs. So by writing unit tests, overall cost of the software is reduced.

Eli Lopian has covered some Unit Testing Myths and Practices on DotNetCurry.com which I recommend reading.

As I have already mentioned, from Visual Studio 2012, Microsoft provided support for third party testing frameworks. PEX framework was available as a power tool. With PEX you could generate test suites with high code coverage. Visual Studio 2015 extends it and provides this feature under the name of Smart Unit Tests.

Smart Unit Tests can be helpful in finding out if the code is testable. It can also help in getting code coverage information like which code paths were visited etc. We need to decide when to fail or pass the test. This can be achieved by adding assertion in the method. We have the liberty of adding as many assertions as needed.

Smart unit test instruments the code under test. The tests are run with a simple input value (depending upon the parameters for the test). The test engine finds out how the input value flows through the code and thus computes the code coverage. There can be various logic branches in the code. If the input value does not reach some paths of the code, then another input value is picked up and then decided if the un-reached branches are getting executed now. Smart unit tests also specify if there are any input values that are not considered as a part of code (for example null value not checked).

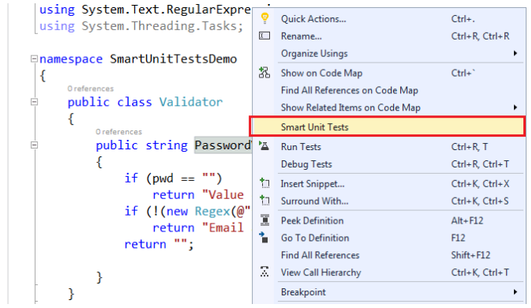
As an illustration of how smart unit testing works, I am going to create a class library using Visual Studio 2015 in which I will add some code and add smart unit tests for the methods in the code.

I have written a method to validate the email address entered. The method takes one parameter of the type *string* which is validated.

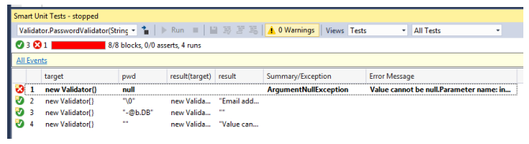
```
public class Validator
{
    public string EmailValidator(string pwd)
    {
        if (pwd == "")
            return "Value cannot be empty";
        if (!(new Regex(@"^[a-zA-Z0-9_\-\.]+@[a-zA-Z0-9_\-\.]+\.[a-zA-Z]{2,}$")).IsMatch(pwd))
            return "Email address is not valid";
        return "";
    }
}
```

Visual Studio 2010 provided the feature of creating unit tests on a method to be tested automatically. In later versions of Visual Studio, this was available as a separate feature from Visual Studio Gallery (but is only used to add references to the class library and provide a test method stub). Now we can generate smart unit tests and execute it even before adding them in a test project.

Right click on the method in Visual Studio 2015 to bring up the option for Smart Unit Tests.



A test suite is generated for the current method. The tests are executed and we get the result. It generates various input values depending upon the parameter(s) required for the method to be tested.

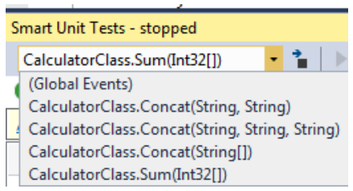


Different input values are created in order to find the flow of all the paths in the method. The number of test data depends upon the types and number of parameters for the method.

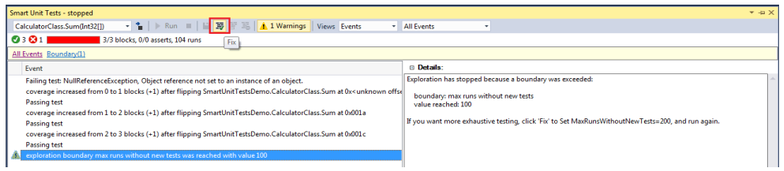
When you run these smart unit tests, you can find out which tests are passing and which are failing.

The option of generating smart unit tests for all the public methods in the class can be availed by selecting the class, and choosing the option of

Smart Unit Tests. It will give us a test suite with all the test methods along with any global events.



Selecting *All Events* will show a list of events. It will also show us if there are any boundary violations. These issues can be fixed on clicking on the Fix button (see icon in the image).



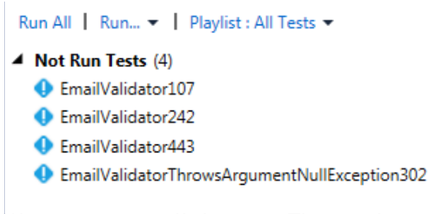
Observe the Fix button provided to fix this issue. This fix will change the count for max runs to 200 as shown in following diagram. This gives us maximum number of consecutive runs without a new test being emitted.

```
[TestClass]
[PexClass(typeof(CalculatorClass))]
[PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException), AcceptExceptionSubtypes = true)]
[PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
public partial class CalculatorClassTest
{
    [PexMethod(MaxRunsWithoutNewTests = 200)]
    public int Sum([PexAssumeUnderTest]CalculatorClass target, int[] numbers)
    {
        int result = target.Sum(numbers);
        return result;
        // TODO: add assertions to method CalculatorClassTest.Sum(CalculatorClass, Int32[])
    }
}
```

We can save these test suites in a test project. Select all the test data and the *Save* button will be enabled.

```
[TestClass]
[PexClass(typeof(Validator))]
[PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException), AcceptExceptionSubtypes = true)]
[PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
public partial class ValidatorTest
{
    [PexMethod]
    public string EmailValidator([PexAssumeUnderTest]Validator target, string pwd)
    {
        string result = target.EmailValidator(pwd);
        return result;
        // TODO: add assertions to method ValidatorTest.EmailValidator(Validator, String)
    }
}
```

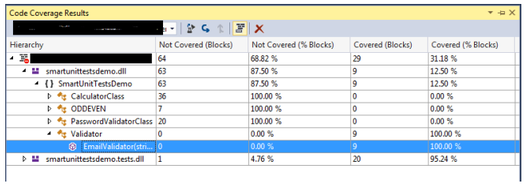
Observe the test project and the method with parameter. *PexMethod* attribute is added to the method along with *TestClass* attribute for the class. In Test Explorer, you can view a list of all the tests added. As usual, build (desktop build) will provide a list of all the test methods added. It can be seen with the help of Test Explorer.



We can execute all the tests. Thus we have created test suite for regression. A file is automatically added to the test project with test data and required assertions in it. If the method code changes, you need to re-run the smart unit test.

You can also find out the code coverage information about the assembly which is instrumented. Select the methods to be executed from Test Explorer, right click and choose the option for *Analyse Code Coverage* for Selected Tests.

Expand the assemblies shown in Code Coverage Results. We can drill down to the code and find out the paths which did not execute. The code paths which executed are shown in blue colour and the others in red colour. With code coverage, we can find out about efficiency of the code.

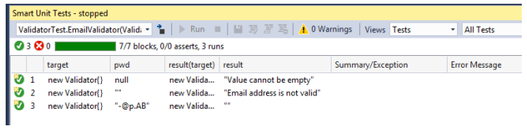


Generated Parameters in Smart Unit Tests

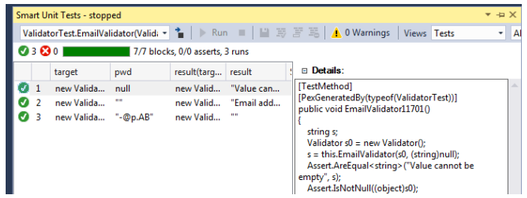
Let us try and find out how the parameters are generated for Smart Unit Test. If you refer to the original values of the parameters, you can observe that the code is not testing for null. The first value for parameter passed is given as null and an exception is thrown.

I changed the equality to null in the code as seen in the following code and now I get a little change in the input values and the results shown.

```
public string EmailValidator(string pwd)
{
    if (pwd == null)
        return "Value cannot be empty";
    if (!(new Regex(@"^[a-zA-Z0-9_\-\.]+@[a-zA-Z0-9_\-\.]+\.[a-zA-Z]{2,}$")).IsMatch(pwd))
        return "Email address is not valid";
    return "";
}
```



The test method generated can be seen in the following image once you select one of the test cases.



The code is run with the simplest value for the input parameter. That is the first value for the parameters generated.

This result shows that it has hit the first *if* statement. The next value for parameter is going to the next *if* statement and finally the third value gives the valid email address. This determines that not only are the random values for parameters generated, but the test engine keeps track of which paths in the code were reached and accordingly generates input values. Any exception encountered is termed as a separate branch for the code. Smart!

In this article, we discussed Smart Unit Tests in Visual Studio 2015. Smart Unit Tests are used for automated white box testing along with code coverage. Smart unit tests are an extension of Pex framework with a lot of enhancements. The usual time taken to release a software is reducing day by day and Smart Unit Tests is one such tool that help us in doing so ■

About the Author

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

Add Notifications in your Website using jQuery

Have you ever seen those notifications on websites that appear at the bottom of the page when you start scrolling down, and disappear when you scroll up? In this article, let's build one for ourselves. We will also see how to give an option to our visitors to permanently close the notification, in case they find it annoying.

Create a new file called 'NotificationsUsingPanels.html'. In terms of HTML, all we need is a simple container to hold our notification message, and a span tag that contains the close button:

```
<div id="note">
    <h3>Your Message Comes Here</h3>
    <span class="close">

    </span>
</div>
```

This Div needs to be positioned in the bottom corner of the page to achieve the effect we're aiming for. We also need the close button at the top-right corner of the Div.

Here's how we can style it to achieve the effect. This css has been saved in css/notificationpanel.css

```
#note{
    display:none;
    position:fixed;
    bottom:0;
    right:3px;
    min-height: 50px;
    height:100px;
    width:500px;
    background: #4679bd;
    padding: 20px 60px;
    font: 24px Georgia, serif;
    color: #fff;
    border-radius: 9px;
}


.close {
    background: transparent url('../
    images/close-sign.png') 0 0
    no-repeat;
    position: absolute;
    top: 5px;
    right: 5px;
    width:40px;
    height:48px;
    display:block;
}
```

We start by setting `position:fixed` to keep our div at the bottom of the page. We have also set visibility of the div to `none`, as our requirement says that the div should be visible *only* when the user scrolls down the page. In order to render the close button, we have explicitly set a width and height for it and have set `display:block`.

This is what we have so far:

To show and hide the notification message when the user scrolls, we will make use of the jQuery `.scroll()` event which fires whenever the element's scroll position changes, regardless of the cause. In our case, we will apply it to the browser window.

```
$(window).scroll(function () {
...
});
```

This is a shortcut for `$(window).on( "scroll", function())`. Here's the entire code:

```
$(function () {
    var $window = $(window),
        $body = $('body'),
        $note = $("#note");

    $window.scroll(function () {
        if ($window.scrollTop() > $body.
height() / 2) {
            $note.fadeIn();
        }
        else {
            $note.fadeOut();
        }
    });
});
```

`scrollTop()` gets the current vertical position of the scroll bar and we compare it with the height of the scroll to see if we are halfway there. If yes, we show the notification using `fadeIn()`. When the user scrolls up, the same logic is used to determine if we are about halfway up, and hide the notification using `fadeOut()`.

In order to close the notification, use the following code:

```
$('.close').on('click', function () {
    $note.fadeOut("slow");
    $window.off('scroll');
});
```

On the click event of the span, we use `fadeOut()` to hide the notification panel and remove the scroll event listener from the window object using `off()`.

View the page in the browser and you will be able to show and hide the notification as you scroll.

See a Live Demo

Using Cookies to hide the notification permanently

Some of you by now might have noticed that this solution is not complete. When you hit that close button, the notification is hidden only until the user decides to reload the page again.

In order to hide the notification permanently, we will need to use a *Cookie*. Cookies are needed because HTTP is stateless. This means that HTTP has no way to keep track of a user's preference on every page reload. This is where a Cookie is useful so that it remembers and saves the user's preference. Let's modify our code to use cookies:

Create a new file called 'NotificationsUsingCookies.html'. Use the same markup as we used earlier. Use the following code:

```
var $window = $(window),
  $body = $('body'),
  $note = $("#note"),
  notifyCookie = document.cookie;

$window.scroll(function () {
  if (($window.scrollTop() > $body.
  height() / 2) &&
    (notifyCookie.search('hideNote') ===
    -1))
    {
      $note.fadeIn();
    }
    else
    {
      $note.fadeOut();
    }
});


$('.close').on('click', function () {
  $note.fadeOut("slow");
  $window.off('scroll');
  document.cookie = 'hideNote=true;
  expires=' +
    (new Date(Date.now() + (10000 * 1)).
    toUTCString());
});
```

The code more or less is the same, except that we are setting a cookie to save the user's preference and reading it while displaying the notification.

For demo purposes, the cookie is set for 10 seconds. However you can always increase the duration to an hour, a day or even a year.

This was a very basic example of using cookies. If you are looking for a complete example of reading/writing cookies with full unicode support, check the Mozilla documentation link in the Further Reading section of this article. Also note that this example will not work if the user decides to delete the cookie or if the browser does not support cookies.

See a Live Demo

Using a Cookie plugin

Ever since the pre HTML5 era, I have always found the behaviour of `document.cookie` to be strange. For eg: when you assign to it, it adds/updates a cookie instead of replacing it. Overall I feel it could have been designed better, than what we have now. I kind of tend to rely on a nice cookie plugin, rather than using cookies directly in my code.

The *jquery-cookie* plugin by Klaus Hartl (https://github.com/carhartl/jquery-cookie/tree/v1.4.1) is a nice lightweight jQuery plugin for reading, writing and deleting cookies. It's maintained regularly, contains a hosts of features, handles browser quirks and is easy to use.

Create a new file called 'NotificationsUsingCookiesPlugin.html'. We will use the same markup as we used earlier.

Add the following reference to the page:

```
<script src="../scripts/jquery.cookie.
js"></script>
```

Let's modify our sample to use the jQuery cookie plugin instead. The code mostly remains the same except for some changes highlighted in **bold**:

```
var $window = $(window),
  $body = $('body'),
  $note = $("#note");

  $window.scroll(function () {
    if (($window.scrollTop() > $body.
    height() / 2) &&
    (typeof $.cookie('hideNote') ===
    "undefined"))
    {
      $note.fadeIn();
    }
    else
    {
      $note.fadeOut();
    }
  });


$('.close').on('click', function () {
  $note.fadeOut("slow");
  $window.off('scroll');
  var date = new Date(Date.now() +
  (30000));
  $.cookie('hideNote', 1, { expires:
  date });
});
```

To make it a session cookie, omit the `expires` option. This way the cookie gets deleted when the browser exits.

If your browser supports HTML5, you can avoid cookies and instead use HTML5 local storage to save user preferences. Unfortunately covering an HTML5 specific example is beyond the scope of this article but you can always read http://www.dotnetcurry.com/showarticle.aspx?ID=1001 to learn more about how to use HTML5 storage.

For demo purposes, the cookie is set for 30 seconds. However you can increase the duration to an hour, a day or even a year.

See a Live Demo

Further Reading:

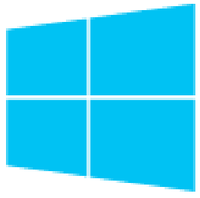
http://api.jquery.com/scroll/
https://developer.mozilla.org/en-US/docs/Web/API/document.cookie ■

About the Author

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

> Cloud computing with Windows Azure provides the capability for developing scalable, durable and highly available applications. For high availability, application data should be always available with scalable data repositories which have the capability of auto-load balancing. Windows Azure Storage APIs make this possible. Windows Azure Storage is a scalable and highly available service for storing any kind of application and non-application data. It has the capability to store and process hundreds of terabytes of data.

Using Azure Storage API in an ASP.NET MVC Application



Windows Azure

Windows Azure Storage services provides some of the following storage options: Tables, Blob and Queue.

Tables are a type of heterogeneous entity container that you can store and retrieve using a key. *Blob* can be used to store files, and you can use *queues* to decouple two applications and enable async communication. Since our application makes use of Table and BLOB, we won't be focusing on Queue in this article. Please visit this link to read about Queue and other kinds of storage.

Table Storage

A Table Storage is a NoSQL key-attribute data store. The data is stored in a structured form and allows rapid development and fast access to a large amount of data. The table storage offers highly available and largely scalable storage. The difference between a Table Storage and a Relational table is, the Table Storage has a schema less design of the data storage. Since it is schema less in nature, the data can be stored as per the needs of the application. For e.g. in a People Survey application, some people may not have a Mobile Number, Email, or a permanent address. Here imagine the pain of designing a relational table where it is always a challenge to choose between Allow NULL or Not NULL columns. The Table storage is a key-attribute store because each value is stored with a type property name, the collection of properties, and values, called as *entity*. The property can further be used for filtering or selecting information from the Table Storage. Each row in the Table Storage has a *RowKey*, which is a unique identification of the entity in the Table Storage. This also has *PartitionKey* which is used to define partition groups in Table Storage.

Blob Storage

A Blob Storage offers a cost-effective and scalable solution to store large amount of unstructured data in the cloud. This data can be in the following form:

- Documents
- Photos, Videos, etc.
- Database backup files or Device backup
- High Definition Images and Videos, etc.

Every blob is organized into a *container*. A storage account can contain any number of containers, which can further contain any number of blobs, up to a 500 TB capacity limit of the storage account. The Blob storage provides two types of blobs - Block blobs and Page blobs (disks). Block blobs are optimized for streaming and storing cloud objects. These are good for storing documents, media files, backups etc. The size can be up to 200 GB. Page blobs are optimized for representing VHDs disks for Azure Virtual Machines, which supports random writes, and its size can be up to 1 TB.

In case you are interested, read a good article by Kunal on Azure Blob Storage Snapshots using Client Library and REST APIs.

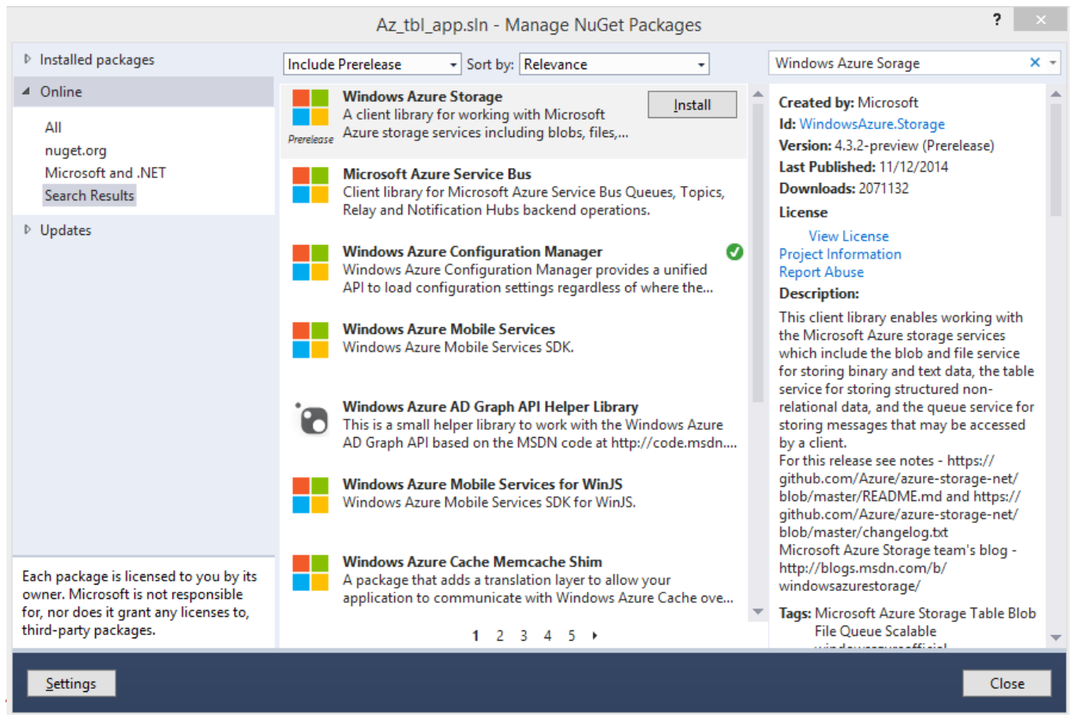
In this article, we will implement an application for managing profiles of Job seekers. The application

will allow a Job seeker to enter his/her basic information after login and also allow to upload a resume. The application will store the profile information in Table Storage, whereas the resume will be stored using BLOB Storage.

Before we get started, we need an active Windows Azure Subscription. Head over to manage.windowsazure.com and create a trial subscription using your Hotmail account. On the portal page, select the Storage Account and create a new storage account. Once the storage account is created, select it and the portal will display the *Manage Access Keys* option at the bottom of the page. Click on it and copy the Storage Name and primary access keys and paste them in a notepad. We will be needing it later.

Step 1: Open Visual Studio. This article code uses VS 2013 and Windows Azure SDK 2.5. Create a blank solution of the name Az_tbl_app. In this solution, add a class library project of the name *EntityStores* and an MVC application of the name *mvcapp*.

Step 2: Since we need Windows Azure Storage APIs for our application, right-click on the solution and from the Manager NuGet Packages > select *Windows Azure Storage* as shown in the following figure:



This will add the necessary libraries in both the projects.

Step 3: In the *EntityStores* project, add a class as shown here:

```
using Microsoft.WindowsAzure.Storage.
Table;
using System.ComponentModel.
DataAnnotations;

namespace EntityStores
{
  public class ProfileEntity :
  TableEntity
  {
    public ProfileEntity()
    {
    }

    public ProfileEntity(int profid,
    string email)
    {
      this.RowKey = profid.ToString();
      this.PartitionKey = email;
    }

    public int ProfileId { get; set; }
    [Required(ErrorMessage=”FullName is
    Must”)]
    public string FullName { get; set; }

    public string Profession { get; set;}
    public string Qualification { get;
    set; }
    [Required(ErrorMessage = “University
    is Must”)]

    public string University { get; set;}
    [Required(ErrorMessage = “ContactNo
    is Must”)]

    public string ContactNo { get; set; }
    [Required(ErrorMessage = “Email is
    Must”)]

    public string Email { get; set; }
    public int YearOfExperience { get;
    set; }

    public string ProfessionalInformation
    { get; set; }
    public string ProfilePath { get; set;
    }

  }
}
```

In this code, the *ProfileEntity* class is derived from the *TableEntity* class. This class defines properties used for storing profile information. Here the ProfileId is the row key and the Email is the partition key which can be further used for data filtering.

Build the project and add its reference in the MVC application, which we created earlier in the same solution.

Step 4: In the MVC Application, open the web.config file of the application and in <appSettings>, add the Storage Account Name and the key as shown in the following code:

```
<add key=”webjobstorage”
value=”DefaultEndpointsProtocol=
https;AccountName=MyAccount;AccountKey=
Key” />
```

I have added placeholders in the AccountName and AccountKey for obvious reasons. Replace them with the ones you have received when you signed up for an Azure subscription.

Step 5: In the Models folder, add a class file that contains the code for working with the Table Storage. This class reads Storage Account information from the web.config file. The code creates an instance of the *CloudStorageAccount* object. This object makes the Storage available to the application. The code also creates an instance of the *CloudTableClient* object. This object represents the Windows Azure Table Storage Service client object using which CRUD Operations can be performed on the table. To perform operations on the Table Storage, we need an instance of the *CloudTable* object, which represents the Windows Azure Table. This contains an *Execute()* method to execute operations on the table. The TableOperation object contains method like the *Insert (), Delete (), Replace ()* and *Retrieve ()* to perform operations on the Windows Azure table. The *TableQuery* object is used to query against the table based upon a condition criteria. The following code shows detailed operations.

```
using EntityStores;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.
Table;
using System.Collections.Generic;
using System.Configuration;

namespace mvcapp.Models
{
  /// <summary>
  /// Interface Containing Operations
  for
  /// 1. Create Entity in Table =>
  CreateEntity
  /// 2. Retrieve Entities Based upon
  the Partition => GetEntities
  /// 3. Get Single Entity based upon
  partition Key and Row Key => GetEntity
  /// </summary>
  public interface ITableOperations
  {
    void CreateEntity(ProfileEntity
    entity);
    List<ProfileEntity>
    GetEntities(string filter);
    ProfileEntity GetEntity(string
    partitionKey, string rowKey);

  }

  public class TableOperations :
  ITableOperations
  {
    //Represent the Cloud Storage
```

```
Account, this will be instantiated
//based on the appsettings
CloudStorageAccount storageAccount;
//The Table Service Client object
used to
//perform operations on the Table
CloudTableClient tableClient;

/// <summary>
/// COnstructor to Create Storage
Account and the Table
/// </summary>
public TableOperations()
{
  //Get the Storage Account from the
  conenction string
  storageAccount =
  CloudStorageAccount.
  Parse(ConfigurationManager.
  AppSettings[“webjobstorage”]);
  //Create a Table Client Object
  tableClient = storageAccount.
  CreateCloudTableClient();

  //Create Table if it does not
  exist
  CloudTable table = tableClient.
  GetTableReference
  (“ProfileEntityTable”);
  table.CreateIfNotExists();
}

/// <summary>
/// Method to Create Entity
/// </summary>
/// <param name=”entity”></param>
public void
CreateEntity(ProfileEntity entity)
{

  CloudTable table = tableClient.
  GetTableReference
  (“ProfileEntityTable”);
  //Create a TableOperation object
  used to insert Entity into Table
  TableOperation insertOperation =
  TableOperation.Insert(entity);
  //Execute an Insert Operation
  table.Execute(insertOperation);
}
/// <summary>
/// Method to retrieve entities
based on the PartitionKey
/// </summary>
/// <param name=”filter”></param>
/// <returns></returns>
public List<ProfileEntity>
GetEntities(string filter)
{
List<ProfileEntity> Profiles = new
List<ProfileEntity>();
CloudTable table = tableClient.
GetTableReference
(“ProfileEntityTable”);

TableQuery<ProfileEntity> query =
new TableQuery<ProfileEntity>()
.Where(TableQuery.
GenerateFilterCondition(“Email”,
QueryComparisons.Equal, filter));


foreach (var item in table.
ExecuteQuery(query))
{
  Profiles.Add(new ProfileEntity()
  {
    ProfileId = item.ProfileId,
    FullName = item.FullName,
    Profession = item.Profession,
    Qualification = item.
    Qualification,
    University = item.University,
    ContactNo = item.ContactNo,
    Email = item.Email,
    YearOfExperience = item.
    YearOfExperience,
    ProfilePath = item.ProfilePath,
    ProfessionalInformation =
      item.ProfessionalInformation
    });
  }

  return Profiles;
}

/// <summary>
/// Method to get specific entity
   based on the Row Key and the
   Partition key
/// </summary>
/// <param name=”partitionKey”>
   </param>
/// <param name=”rowKey”></param>
/// <returns></returns>
public ProfileEntity GetEntity(string
partitionKey, string rowKey)
{
  ProfileEntity entity = null;

  CloudTable table = tableClient.
  GetTableReference
  (“ProfileEntityTable”);

  TableOperation tableOperation =
  TableOperation.Retrieve
  <ProfileEntity>(partitionKey,
  rowKey);
      entity = table.
      Execute(tableOperation).Result as
      ProfileEntity;

      return entity;
    }

  }
}
```

Step 6: In the Models folder, add a new class file where we will implement the code for performing BLOB operations. This class will be used to upload the Profile file of the end-user to the BLOB storage. This class makes use of the *CloudBlobContainer* class used to create the BLOB container. This container will act as a repository for all blobs uploaded. The *CloudBlobClient* object represents the Cloud Blob Storage service object using which requests against the BLOB service can be executed. The *CloudBlockBlob* represents the Blob to be uploaded as a set of blocks.

```
using System;
using System.Web;

using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.
Blob;
using System.Configuration;
using System.Threading.Tasks;
using System.IO;


namespace mvcapp.Models
{

  /// <summary>
  /// Class to Store BLOB Info
  /// </summary>



  /// <summary>
  /// Class to Work with Blob
  /// </summary>
  public class BlobOperations
  {
    private static CloudBlobContainer
    profileBlobContainer;

    /// <summary>
    /// Initialize BLOB and Queue Here
    /// </summary>
    public BlobOperations()
    {
     var storageAccount =
     CloudStorageAccount.
     Parse(ConfigurationManager.
     AppSettings[“webjobstorage”].
     ToString());

     CloudBlobClient blobClient =
     storageAccount.
     CreateCloudBlobClient();

     // Get the blob container
     reference.
     profileBlobContainer = blobClient.
     GetContainerReference
     (“profiles”);

     //Create Blob Container if not
       exist
     profileBlobContainer.
     CreateIfNotExists();
  }


/// <summary>
/// Method to Upload the BLOB
/// </summary>
/// <param name=”profileFile”>
    </param>
/// <returns></returns>

    public async Task
    <CloudBlockBlob>UploadBlob
    (HttpPostedFileBase profileFile)
    {
      string blobName = Guid.
      NewGuid().ToString()
      + Path.GetExtension(profileFile.
      FileName);

      // GET a blob reference.
      CloudBlockBlob profileBlob =
      profileBlobContainer.
      GetBlockBlobReference(blobName);

      // Uploading a local file and
         Create the blob.
      using (var fs = profileFile.
      InputStream)
      {
        await profileBlob.
        UploadFromStreamAsync(fs);
      }
      return profileBlob;
    }

  }
}
```

Step 7: In the Controllers folder, add a new controller of the name ProfileManagerController. This will contain the Index() and Create() HttpGet and HttpPost action methods.

```
using System;
using System.Collections.Generic;
using System.Web;
using System.Web.Mvc;

using EntityStores;
using mvcapp.Models;
using System.Threading.Tasks;
using Microsoft.WindowsAzure.Storage.
Blob;

namespace mvcapp.Controllers
{
  [Authorize]
  public class ProfileManagerController :
  Controller
  {
    BlobOperations blobOperations;
    TableOperations tableOperations;
    public ProfileManagerController()
    {
      blobOperations = new
      BlobOperations();
      tableOperations = new
      TableOperations();
    }
    // GET: ProfileManager
    public ActionResult Index()
    {
      var profiles = tableOperations.
      GetEntities(User.Identity.Name);
      return View(profiles);
    }

    public ActionResult Create()
    {
      var Profile = new ProfileEntity();
      Profile.ProfileId = new Random().
      Next(); //Generate the Profile Id
      Randomly
      Profile.Email = User.Identity.Name;

      // The Login Email
      ViewBag.Profession = new
      SelectList(new List<string>()
      {
        "Fresher","IT","Computer
        Hardware","Teacher","Doctor"
      });

      ViewBag.Qualification = new
      SelectList(new List<string>()
      {
        "Secondary","Higher
        Secondary","Graduate","Post-
        Graduate","P.HD"
      });
      return View(Profile);
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult>
    Create(
      ProfileEntity obj,
      HttpPostedFileBase profileFile
    )
    {

      CloudBlockBlob profileBlob = null;
      #region Upload File In Blob
      Storage
      //Step 1: Uploaded File in BLob
      Storage
      if (profileFile != null &&
      profileFile.ContentLength != 0)
      {
        profileBlob = await
        blobOperations.UploadBlob
        (profileFile);
        obj.ProfilePath = profileBlob.Uri.
        ToString();
      }
      //Ends Here
      #endregion

      #region Save Information in Table
      Storage
      //Step 2: Save the Information in
      the Table Storage

      //Get the Original File Size
      obj.Email = User.Identity.Name; //
      The Login Email
      obj.RowKey = obj.ProfileId.
      ToString();
      obj.PartitionKey = obj.Email;

      //Save the File in the Table
      tableOperations.CreateEntity(obj);
      //Ends Here
      #endregion

      return RedirectToAction("Index");
    }
  }
}
```

The *ProfileManagerController* is decorated with the [Authorize] filter. This indicates that the end-user must login before making a call to this controller.

The Constructor of the controller creates objects of *TableOperations* and *BlobOperation* class. These will create the Table and BLOB storage. The *Index* action method retrieves profile information based on the UserName. The Create() HttpGet action method generates the ProfileId using *Random* class. This also defines *ViewBag* for Profession and Qualification collection so that they can be passed to the Create view during scaffolding. The Create HttpPost action method accepts *ProfileEntity* instance and the *HttpPostedFileBase* object to upload the file. This action method contains code in two steps. Step 1 uploads the file in BLOB and Step 2 stores profile information in the Table storage. Scaffold the Index and Create views from the above controller. Make the following changes in the Create View, (yellow marked)

```
@using (Html.BeginForm("Create",
"ProfileManager", FormMethod.Post, new {
enctype = "multipart/form-data" }))
{
  @Html.AntiForgeryToken()

  <div class="form-horizontal">
  <h4>ProfileEntity</h4>
  <hr />
  @Html.ValidationSummary(true, "", new
  { @class = "text-danger" })

  <div class="form-group">
  @Html.LabelFor(model => model.
  ProfileId, htmlAttributes: new { @class
  = "control-label col-md-2" })

    <div class="col-md-10">
      @Html.EditorFor(model => model.
      ProfileId, new { htmlAttributes =
      new { @class = "form-control" } })
      @Html.ValidationMessageFor(model
      => model.ProfileId, "",
      new { @class = "text-danger" })
    </div>
  </div>

  <div class="form-group">
    @Html.LabelFor(model => model.
    FullName, htmlAttributes: new { @
    class = "control-label col-md-2" })

    <div class="col-md-10">
      @Html.EditorFor(model => model.
      FullName, new { htmlAttributes =
      new { @class = "form-control" } })
      @Html.ValidationMessageFor(model
      => model.FullName, "",
      new { @class = "text-danger" })
    </div>
  </div>

  <div class="form-group">
    @Html.LabelFor(model => model.
    Profession, htmlAttributes: new { @
    class = "control-label col-md-2" })

    <div class="col-md-10">
      @Html.DropDownList("Profession")
      @Html.ValidationMessageFor(model
      => model.Profession, "",
      new { @ class = "text-danger" })
    </div>
  </div>

  <div class="form-group">
    @Html.LabelFor(model => model.
    Qualification, htmlAttributes:
    new { @class = "control-label col-
    md-2" })

    <div class="col-md-10">
      @Html.DropDownList
      ("Qualification")
      @Html.ValidationMessageFor(model
      => model.Qualification, "",
      new { @class = "text-danger" })
    </div>

  <div class="form-group">
    @Html.LabelFor(model => model.
    University, htmlAttributes:
    new { @class = "control-label col-
    md-2" })

    <div class="col-md-10">
      @Html.EditorFor(model => model.
      University, new { htmlAttributes
      = new { @class = "form-control"
      } })

      @Html.ValidationMessageFor(model
      => model.University, "",
      new { @class = "text-danger" })
    </div>
  </div>

  <div class="form-group">
    @Html.LabelFor(model => model.
    ContactNo, htmlAttributes:
    new { @class = "control-label col-
    md-2" })

    <div class="col-md-10">
      @Html.EditorFor(model => model.
```

```
    ContactNo, new { htmlAttributes
    = new { @class = “form-control”
    } })

    @Html.ValidationMessageFor(model
    => model.ContactNo, “”,
    new { @class = “text-danger” })
  </div>
</div>

<div class=”form-group”>
  @Html.LabelFor(model => model.
  YearOfExperience, htmlAttributes:
  new { @class = “control-label col-
  md-2” })

  <div class=”col-md-10”>
    @Html.EditorFor(model => model.
    YearOfExperience, new {
    htmlAttributes = new { @class =
    “form-control” } })
    @Html.ValidationMessageFor(model
    => model.YearOfExperience, “”,
    new { @class = “text-danger” })
  </div>
</div>

<div class=”form-group”>
  @Html.LabelFor(model => model.
  ProfessionalInformation,
  htmlAttributes: new { @class =
  “control-label col-md-2” })

   <div class=”col-md-10”>
     @Html.EditorFor(model =>
     model.ProfessionalInformation,
     new { htmlAttributes = new { @
     class = “form-control” } })
     @Html.
     ValidationMessageFor(model =>
     model.ProfessionalInformation,
     “”, new { @class = “text-
     danger” })
   </div>
</div>

<div class=”form-group”>
  <label class=”control-label col-
  md-2” for=”profileFile”>Upload
  Profile File</label>
  <div class=”col-md-10”>
    <input type=”file”
    name=”profileFile” accept=”doc/*”
    class=”form-control fileupload”/>
  </div>
</div>

<div class=”form-group”>
  <div class=”col-md-offset-2 col-
      md-10”>
      <input type=”submit” value=”Create”
      class=”btn btn-default” />
    </div>
  </div>
</div>
}
```

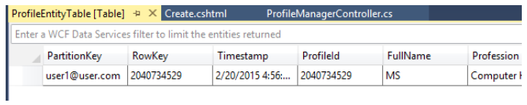
Since we are uploading the file using this view, the Html.BeginForm method contains parameters for FormMethod.Post and enctype=”multipart/form-data”.

Step 8: Run the application and create a user for the application e.g. user1@user.com

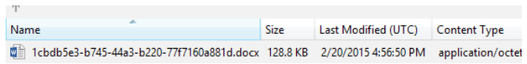
Now navigate to the Create View and enter the Profile Information as shown here:

Click on the *Create* button and the data will be saved. To verify the operation, in the Visual Studio Server Explorer expand the Table and BLOB Storage and you will find the following information:

The above figure shows the profiles *Blob* and *ProfileEntityTable*. The data can be seen from the Table Storage as shown here:



Similarly we can see the uploaded BLOB as shown here:



Conclusion:

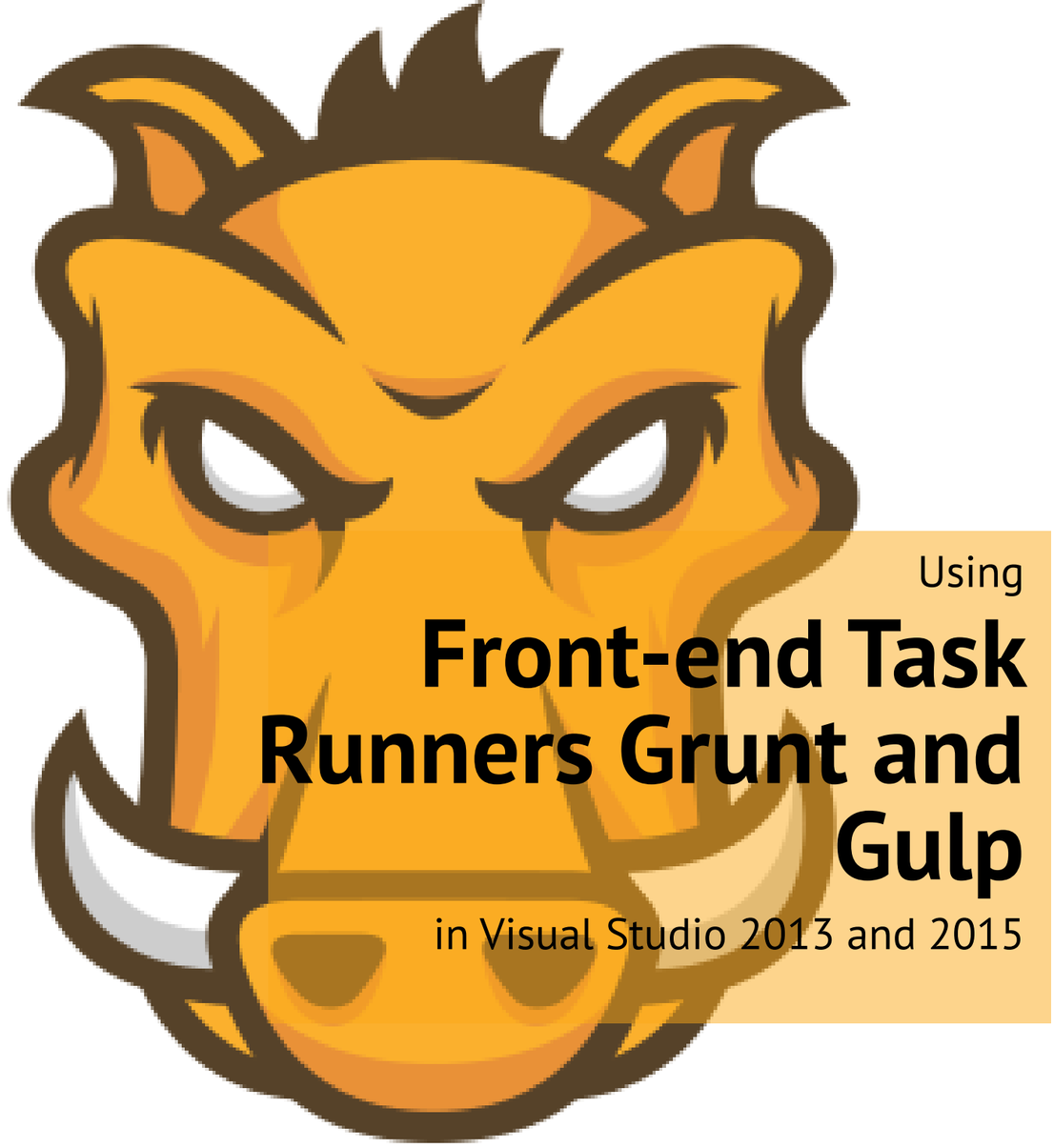
Windows Azure Storage provides highly scalable and available storage for applications. The schema less Table storage allows us to store application data whereas BLOB storage provides the file repositories for the application. This eliminates the need for complex Sql Server Relational tables and the Image column in it.

In one of our future articles, we will see how to compress BLOB before storing it ■

Download the entire source code from our GitHub Repository at bit.ly/dncm17-azurestoragemvc

About the Author

Mahesh Sabnis is a Microsoft Certified Trainer (MCT) since 2005 and has conducted various Corporate Training programs for .NET Technologies (all versions). Follow him on twitter @maheshdotnet. Mahesh blogs regularly on .NET Server-side & other client-side Technologies at bit.ly/HsS2on



Using

Front-end Task Runners Grunt and Gulp

in Visual Studio 2013 and 2015

Though Visual Studio started as an IDE for building applications based on Microsoft Technologies, lately the IDE is expanding its domain by adding capabilities to build apps based on other technologies too, including many open source technologies. Projects like Node.js Tools for Visual Studio (NTVS) and Python Tools for Visual Studio (PTVS) prove the openness the IDE is adopting these days.

Front-end web development went through a number of changes in the recent past. One primary change especially when it comes to developing on the Microsoft stack has been to take advantage of the rich ecosystem of tools created by the community. At times, it feels like front-end web development was always incomplete without tools like Bower, Grunt and Gulp - some JavaScript based task runners and package manager. It would be hard to find an open source front-end project on GitHub that doesn't use these tools. So any rich web development environment not taking advantage of this already existing rich ecosystem, may sound a bit dated. Being a fan of Visual Studio for years, I was expecting these features to be supported by the IDE. Thankfully, the IDE got a couple of extensions recently that fills this gap for us.

In this article, we will see how these tools can be used to build modern web apps on Visual Studio.

Installing the Visual Studio Extensions

If you are a user of Visual Studio for a long time, you would be using Web Essentials too. Web Essentials is an extension from Microsoft created by Mads Kristensen, that adds a lot of features to make web development better on Visual Studio. In Visual Studio 2015, Web Essentials adds the features of adding Grunt and Gulp to Visual Studio. For VS 2013, we have a plugin called Grunt Launcher, written by a community member to add tooling support for Grunt and Gulp.

Before you move forward, make sure that you have these extensions installed. They can be installed either directly from Visual Studio menu (Tools > Extensions and Updates), or by downloading the vsix files from Visual Studio Extensions Gallery.

- For VS 2013: Grunt Launcher
- For VS 2015: Web Essentials

You need to restart Visual Studio in order to get these extensions working.

An Introduction to the Tools

The tools that I have been mentioning till now, namely Bower, Grunt and Gulp, work on top of Node.js. You should have Node.js already installed on your system. The three tools we are going to work on can be installed using NPM (Node Package Manager). Following are the commands and a brief note on these tools:

- **bower**: Bower is a package manager used to manage front-end packages in a project. It can be used to fetch packages from any Git repository. By default, it uses GitHub as the source of repositories. Bower has to be installed globally using the following command:

```
npm install bower -g
```

- **grunt**: Grunt is a task runner. It can be used to perform a series of operations like concatenation, minification, copy, clear, check for quality using JS Hint, start an express server and similar tasks. Like bower, grunt also needs to be installed globally. The list of tasks have to be configured on a special file named Gruntfile.js. This name is a convention and shouldn't be changed. The configuration has to be defined using an object literal. Grunt has to be installed globally using the following command:

```
npm install grunt -g
```

- **gulp**: Gulp is an alternative to Grunt. Unlike Grunt, tasks for gulp have to be defined using pipes and streams. The tasks have to be defined in a file named Gulpfile.js; like Grunt, this file has to be named by convention. Gulp has to be installed globally as well using the following command:

```
npm install gulp -g
```

Note: *To use Grunt and Gulp in VS 2015 Preview/ CTP and in VS 2013, you need to have them installed globally. Final release of VS 2015 will use the packages installed in the project and won't need the global packages.*

Now that you got some idea on the tools to be used, let's use them in an application.

About the Demo Application

We will not be building the application from scratch. Instead, I will add grunt and gulp support to an existing application. The demo application is a Movie List application (If you read my article on Node.js tools on Visual Studio, I am using the same demo here). It is a simple movie list application that has the following features:

- Shows a list of movies
- Adding a new movie
- Marking a movie as released
- Marking a movie as watched

If you want to follow along this article, download the sample code of this article and open the MovieList – 2013 or, MovieList – 2015 on either VS 2013 or VS 2015 respectively. These folders contain all the required code, but don't have NPM packages, Bower packages and Grunt or Gulp added to them. VS 2013 project is created from an empty ASP.NET template and the VS 2015 project is created using ASP.NET MVC 5 starter template. The reason for choosing two different templates is to show that Grunt and Gulp can be used with any kind of application.

Both these projects contain Entity Framework code first classes and ASP.NET Web API or MVC 6 controllers to serve data. On the front-end, they have just one view built using Bootstrap and Angular. This view fetches data from the API and binds it to the view. You won't be able to run the application unless you have the required scripts and styles at their target places.

This view fetches data from the API and binds it to the view. You won't be able to run the application unless you have the required scripts and styles at their target places.

In rest of the article, we will add the following Grunt or Gulp tasks to this application:

- Install and Copy bower components
- Run Karma tests on Chrome
- Concatenate JavaScript files
- Copy concatenated JavaScript file and CSS file to a target folder
- Clear contents of a folder

Using Grunt and Bower on Visual Studio

Once you open the project inside Visual Studio, check the files and folders to get familiar with the code and the way it is organized. The project was created using ASP.NET Empty project template and then I added some other files to it. As you see, the project doesn't have any existing Node and bower dependencies. Let's add them now.

Add a new file to the project and name it package.json. This file will contain the list of Node.js dependencies required by the project. Add the following content to the file:

```
{
  "name": "Movie-list",
  "preferGlobal": true,
  "version": "0.1.0",
  "author": "Ravi Kiran",
  "description": "A sample movie list
  application",
  "bin": {
    "package-name": "./bin/package-name"
  },
  "dependencies": {
  },
  "analyze": false,
  "devDependencies": {
    "grun": "^0.4.5",
    "grunt-bowercopy": "^1.2.0",
    "karma": "^0.12.31",
    "karma-jasmine": "^0.3.5",
    "grunt-karma": "^0.10.1",
    "grunt-contrib-copy": "^0.7.0",
    "grunt-contrib-concat": "^0.5.0",
    "grunt-contrib-clean": "^0.6.0"
  },
  "license": "MIT",
  "engines": {
    "node": ">=0.6"
  }
}
```

If you type-in the dependencies, the IDE shows a list of suggestions based on the text entered. It fetches this information from the global NPM registry. It also shows suggestions for versions. Figure-1 shows a screenshot of the intellisense:

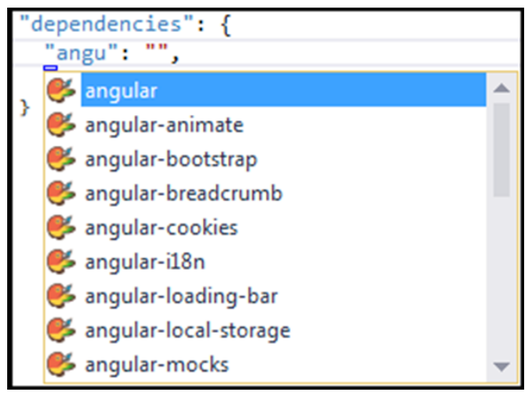


Figure 1: npm intellisense

To install these packages, you can either run the

following command in a command prompt:

```
npm install
```

Or, you can use menu options in Visual Studio. The options are different in VS 2013 and VS 2015. In VS 2013, this option is available on right clicking the package.json file, as shown in Figure-2:

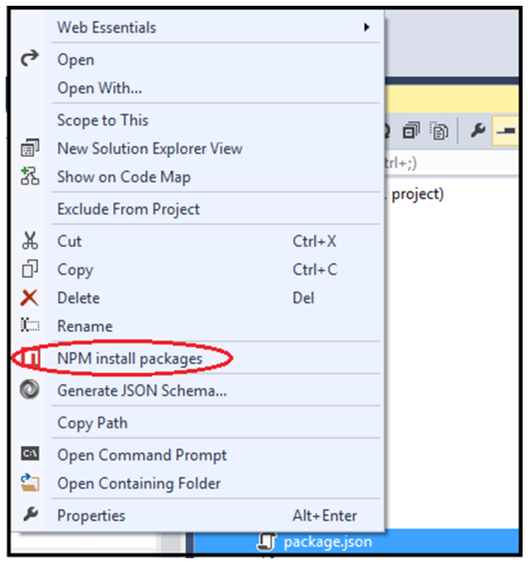


Figure 2: NPM packages in Visual Studio 2013

In VS 2015, this option is available through context menu on NPM node under Dependencies node. Figure -3 shows it:

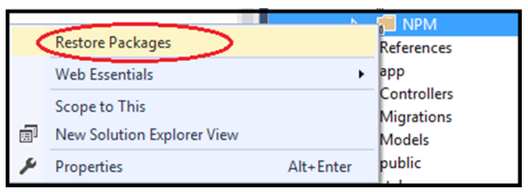


Figure 3: NPM packages in Visual Studio 2015

Now add another file to the project and name it bower.json. It should have following content:

```
{
  “name”: “MovieList”,
  “version”: “0.1.0”,
  “authors”: [
    “Ravi Kiran”
  ],
  “description”: “A sample movie list
  app”,
  “license”: “MIT”,
  “private”: true,
  “ignore”: [
    “**/.*”,
    “node_modules”,
    “bower_components”,
    “test”,
    “tests”
  ],
  “dependencies”: {
    “angular”: “~1.3.11”,
    “bootstrap”: “3.3.2”
  }
}
```

Even in this file, you will get intellisense when you type-in the dependencies.

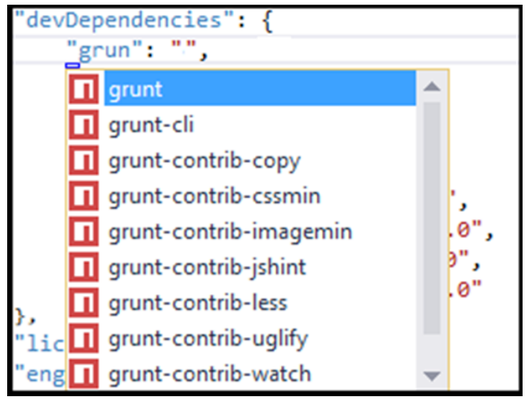


Figure 4: Dependencies Intellisense in bower.json

VS 2013 doesn't have a menu to install Bower components, but VS 2015 has it. In VS 2015, you can install the Bower packages using *Restore Packages* option in the context menu in *Bower* option. I prefer installing the Bower packages using Grunt instead. Right click on the application and choose Add > New Item. From the menu, choose Gruntfile.js if available, otherwise add a new JavaScript file and name it Gruntfile.js. Add the following content to this file:

```
module.exports = function (grunt) {
  ‘use strict’;
  grunt.initConfig({
    // read in the project settings
    from the package.json file into the
    pkg property
    pkg: grunt.file.readJSON(‘package.
    json’)
  });

  //Add all plugins that your project
  needs here
  grunt.loadNpmTasks(‘grunt-bowercopy’);
  grunt.loadNpmTasks(‘grunt-karma’);
  grunt.loadNpmTasks(‘grunt-contrib-
  copy’);
  grunt.loadNpmTasks(‘grunt-contrib-
  concat’);
  grunt.loadNpmTasks(‘grunt-contrib-
  clean’);
};
```

Here we are loading all available grunt tasks and storing the contents of the package.json file in an object for further use.

To install Bower components using Grunt, we need to use the *grunt-bowercopy* task referred above. At times, we may not want to have all Bower dependencies in the default folder. In such cases, we can move these files to a place of our choice by setting paths for each of the dependency. Following snippet shows the Grunt configuration of this task, it has to be added as a property to the object passed to *grunt.initConfig* method in the snippet above:

```
bowercopy: {
  options: {
    runBower: true,
    destPrefix: ‘public/libs’
  },
  libs: {
    files: {
      ‘angular’: ‘angular’,
      ‘jquery’: ‘jquery/dist’,
      ‘bootstrap’: ‘bootstrap/dist/
      css’
    }
  }
}
```

Note: *In VS 2015, modify the value of destPrefix to wwwroot/lib, as all static files are referred from wwwroot by default*

This task will copy the libraries into the folder *libs* under *public* folder.

Though we create a number of files while writing code, we ship just one file containing all the code in a minified format to optimize the network calls by browsers. For the movie list application, I created 3 different JavaScript files. Let's write Grunt tasks to concatenate these files. The files can be minified using grunt-contrib-uglify task, I am leaving it as an assignment to the reader. Alternatively you can also read Using Grunt.js to Merge and Minify JavaScript files in an ASP.NET MVC Application.

Let's concatenate the files using the Grunt task grunt-contrib-concat. Following is the Grunt configuration for this task, it concatenates the three files and stores the resultant file in another folder.

```
concat: {

  options: {
    separator: ‘;’
  },
  dist: {
    src: [‘app/moviesApp.js’, ‘app/
    moviesCRUDSvc.js’, ‘app/MoviesCtrl.
    js’],
    dest: ‘app/combined/moviesCombined.
    js’
  }
}
```

It is a good practice to have all deliverables at one place to make the deployment easier. So let's copy the combined JavaScript file and the CSS file to a single folder. It is done using the grunt-contrib-copy task. Following is the configuration:

```
copy: {
  main: {
    expand: true,
    flatten: true,
    filter: ‘isFile’,
    src: [‘app/combined/*.js’,’styles/*.
    css’],
    dest: ‘public/dist/’
  }
}
```

Note: *In VS 2015, modify the value of destPrefix to wwwroot/dist*

The option *flatten: true* is used to avoid creation of folder structure under the target folder.

After copying the combined JavaScript file, we can remove it from the place where it was created. Following is the *clean* task that does this:

```
clean: [“app/combined/”]
```

We need these tasks to run sequentially. So let’s combine them in the right order into one task. Following is the task:

```
grunt.registerTask(‘default’,
[‘bowercopy:libs’,
‘concat:dist’,’copy:main’, ‘clean’]);
```

Here, *default* is the alias task that runs a number of other tasks internally.
Finally, to run the unit tests using Karma, we need to add a task. Following is the task for it:

```
karma: {
  unit: {
    configFile: ‘karma.conf.js’
  }
}
```

Now we are done with configuring the tasks. Let’s run the tasks and see how it behaves. To run the task, open Task Explorer in Visual Studio through View > Other Windows > Task Explorer.

As you see, the task explorer is smart enough to detect the Grunt file and list all tasks and alias tasks configured. Right click on the default task and click run. This will run all the tasks under default, which means, it will run bower, copy libraries, concatenate, and copy the files to the specified location. Figure-5 shows output of an instance of the execution:

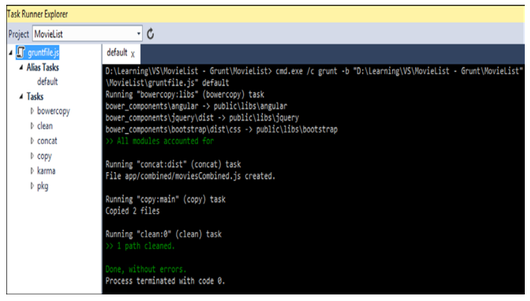


Figure 5: Task Runner Explorer

Now run the application using F5 or Ctrl+F5. You will be able to see the page loading in the browser.

As we didn’t include karma in the default task, it didn’t run tests. We can run karma using the karma node in the Task Runner Explorer. Figure-6 shows an instance of running karma:

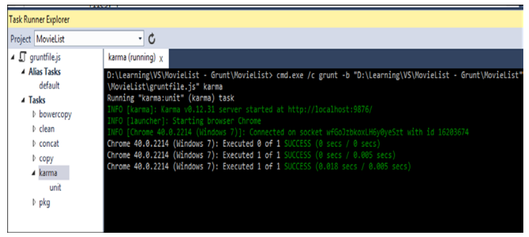


Figure 6: Karma Unit tests

Using Gulp and Bower in Visual Studio

Gulp is an alternative to Grunt. It uses a different approach to solve the same problem. As already mentioned, Gulp uses node streams. Each task in Gulp produces a stream that can be piped into another task to continue execution. This approach is comparatively efficient than the approach used by Grunt, as it doesn’t need to store results physically to make it available to the next task.

Also, being programmers we prefer calling functions over writing configuration objects. In Gulp, we write chained function calls (as we do in LINQ).

Contents of the *bower.json* file would remain the same as in case of Grunt. Contents of *package.json* would change to include Gulp task packages. Following are contents of the *package.json* file:

```
{
  “name”: “Movie-list”,
  “preferGlobal”: true,
  “version”: “0.1.0”,
  “author”: “Ravi Kiran”,
  “description”: “A sample movie list
  application”,
  “bin”: {
    “package-name”: “./bin/package-name”
  },
  “dependencies”: {
  },
  “analyze”: false,
  “devDependencies”: {
    “karma”: “^0.12.31”,
    “karma-jasmine”: “^0.3.5”,
    “gulp”: “^3.8.10”,
    “gulp-bower”: “^0.0.10”,
    “gulp-concat”: “^2.4.3”,
    “gulp-karma”: “^0.0.4”
  },
  “license”: “MIT”,
  “engines”: {
    “node”: “>=0.6”
  }
}
```

Install these packages using the Visual Studio context menu options discussed in the previous section.

Add a new JavaScript file to the project and name it Gulpfile.js. This name is a convention, we can’t change the name. To start with, load all required NPM packages to this file:

```
var gulp = require(‘gulp’);
var bower = require(‘gulp-bower’);
var concat = require(‘gulp-concat’);
var karma = require(‘gulp-karma’);
```

Let’s install bower packages using the gulp-bower package. Following snippet creates a gulp task using this package to install all bower components inside bower_components folder:

```
gulp.task(‘bower’, function () {
    return bower(‘./bower_components’);
});
```

Now that we have the front-end libraries, let’s move them to a different folder. For this, we need to set a destination for each of them. Following task does it:

```
gulp.task(‘copyLibs’, [‘bower’],
function () {
  gulp.src([‘bower_components/
  angular/*.*’])
    .pipe(gulp.dest(‘public/libs/
    angular’));

  gulp.src([‘bower_components/bootstrap/
  dist/css/*.*’])
    .pipe(gulp.dest(‘public/libs/
    bootstrap’));

  gulp.src([‘bower_components/jquery/
  dist/*.*’])
    .pipe(gulp.dest(‘public/libs/
    jquery’));
});
```

Note: *In VS 2015, modify values passed into dest as wwwroot/lib, as all static files are referred from wwwroot by default.*

As you see, this setup is quite different from the setup we did in grunt. It is also a bit tricky to think in the terms of Gulp if you are used to Grunt for a long time. In the above snippet, *gulp.src* is a task and its result is passed to *gulp.dest* using pipe. As mentioned earlier, Gulp works on streams. *gulp.src* produces a stream and this stream is used by *gulp.dest*.

The second parameter in the above snippet is a list of tasks that have to run before this task runs. So whenever we run the *copyLibs* task, it internally runs the *bower* task.

Let’s combine the script files into one file using the *gulp-concat* task. As we can combine multiple tasks using pipes, we can copy the concatenated files to a target folder in the same task. Following is the task:

```
gulp.task(‘concat’, function () {
  return gulp.src([‘app/moviesApp.js’,
  ‘app/moviesCRUDSvc.js’,
  ‘app/MoviesCtrl.js’])

  .pipe(concat(‘public/dist/
  moviesCombined.js’))
  .pipe(gulp.dest(‘.’));
});
```

The only independent task left is copying CSS file to the dist folder. This task is straight forward.

```
gulp.task(‘copyCss’, function () {
  return gulp.src([‘styles/*.css’])
    .pipe(gulp.dest(‘public/dist’));
});
```

Note: *In VS 2015, modify values passed into dest as wwwroot/dist, as all static files are referred from wwwroot by default*

Finally, we need to combine these tasks into one task. Following is the default task that combines all of these tasks:

```
gulp.task(‘default’, [‘copyLibs’,
‘copyCss’, ‘concat’]);
```

To ensure correctness of the code, we should run unit tests too. Following task creates a task to run tests using karma:

```
gulp.task('karma', function () {
  return gulp.src(['tests/*.js']).
  pipe(karma({
    configFile: 'karma.conf.js',
    action: 'watch'
  }));
});
```

Task Runner Explorer in Visual Studio parses the Gulpfile.js file and lists all of the tasks configured as nodes using which we can directly run the tasks. Following figures 7 and 8 show instances of running the default task and the karma unit task

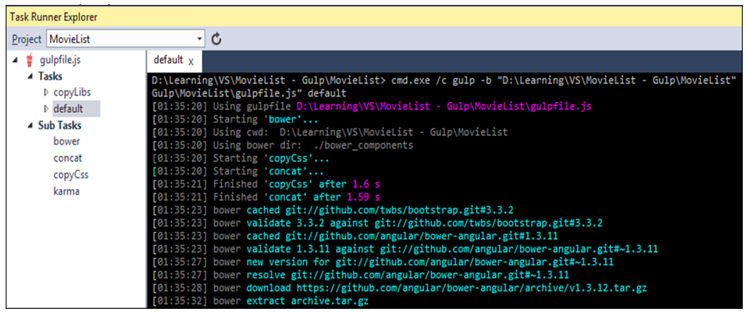


Figure 7: Running Default task

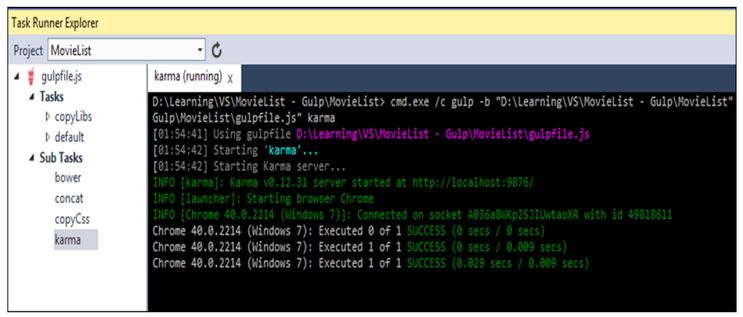


Figure 8: Running Karma Unit task

Conclusion

As you saw, Visual Studio has first class support for working with the front-end task runners. If you have kept yourself away from these tools till now, now is the time to start using them to get your job done and have an experience with the tools that everyone else is using. You will love it!■

Download the entire source code from our GitHub Repository at bit.ly/dncm17-gruntgulpvs

About the Author

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