ISSUE 16 | JAN - FEB 2015

DNCMAGAZINE

www.dotnetcurry.com

Exploring
ServiceStack

VISUAL STUDIO ONLINE WITH **GIT**

Getting Started with **HTML 5** Web Components

ECMA Script 6
New language improvements in JavaScript

Authoring your
First jQuery Plugin

Moving forward with
.NET
Design Patterns

SOFTWARE GARDENING
Agile is not for the faint of heart
SOIL NUTRIENTS

SharePoint as a Content Management System?

CONTENTS

EDITORIAL

Happy New Year Readers! 2014 was a fast paced year for developers working on the Microsoft Stack. New releases in .NET, Visual Studio, Web, Cloud, Phone, Office, SQL Server, and Windows technologies kept everyone busy and excited! On top of that, Microsoft's decision of open sourcing the .NET Core Runtime and libraries and providing a free full version of Visual Studio for the community was the perfect way to end the year.

To warm up during this cold season, we have some cutting edge content for you. Gil gives us an overview of HTML5 Web Components and also demonstrates how to create our own component. In this edition's featured article, Ravi talks about EcmaScript 6 with some new language improvements in JavaScript. Subodh gives us an overview of how TFS and Visual Studio Online have embraced Git as one of the options for version control.

We welcome our new author and SharePoint veteran Todd Crenshaw with a wonderful article on the strengths and drawbacks of SharePoint as a Content Management System. Raj takes a fresh look at .NET Design Patterns and demonstrates very helpful answers to some very common problems.

In our regular Software Gardening column, Craig analyzes Agile to see what it really means. To round off, Ravi explores ServiceStack further and I demonstrate how to create your own jQuery plugin using the Boilerplate plugin template.

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

Suprotim Agarwal

Editor in Chief

CREDITS

Editor In Chief Suprotim Agarwal
suprotimagarwal@a2zknowledgevisuals.com

Art Director Minal Agarwal
minalagarwal@a2zknowledgevisuals.com

Contributing Writers Craig Berntson, Gil Fink, Raj Aththanayake, Ravi Kiran, Subodh Sohoni, Suprotim Agarwal, Todd Crenshaw

Reviewers Alfredo Bardem, Suprotim Agarwal

Next Edition 2nd March 2015
www.dncmagazine.com

POWERED BY
A2Z Knowledge Visuals

ECMA Script 6

New language improvements in JavaScript

If you are a Web 2.0 developer, probably the silliest question someone can ask you today would be "Do you know JavaScript?" JavaScript is an integral part of web development, although the usage of the JavaScript language has changed over the years. The same language that was used a couple of years ago for simple DOM manipulations on the browser, is now used to create extremely rich UI experiences for almost any device, and also to build lightweight web server apps. Today, the day jobs of a sizeable number of developers on this planet is writing only JavaScript.

Though usage of the language has widened, it was not designed with such wide usage in mind. Despite the wide adoption, the language lacks some of the much needed features that other programming languages possess. The JavaScript Community has been very active in developing a number of wrappers and libraries to make the experience better for every developer out there. Thanks to the openness of the web standards; the Web standards committee has been accepting ideas from popular libraries and frameworks and making it a part of the web.

ECMA Script (a.k.a ES) is the specification of JavaScript. Anything that has to become a part of the language has to be specified in ES. The current version of JavaScript used in all popular browsers is based on the ES5 specification. Currently, the ES team is working on standardizing specifications for the next version, i.e. ECMA Script 6 (ES6). This version comes with a number of additions to the syntax, new APIs and improvements to some existing APIs as well. A complete list of features can be found on the Official site for ES6. An HTML version of the specification can also be found on Mozilla's site.

After taking a look at the current version of the specification, I personally feel that the resulting language would make the job of JavaScript developers much easier. Those new to the language or coming from different programming backgrounds will also find useful features and techniques they are already familiar with. Most of the new features are adopted from JavaScript wrappers like TypeScript, CoffeeScript and also from some of the popular libraries that are helping web developers today, to get their job done.

Though it will take some more time to get the final specification of ES6, browsers have started implementing these features and we also have a number of transpilers and polyfills to work with ES6 today. Transpilers and polyfills are tools to try out tomorrow's features, today.

How to write and test ES6 today?

Though ES6 is still in works, we have a good number of options around to try, see and use ES6. Most of the leading browsers have already started implementing features of ES6.

In current versions of Opera and Chrome, you need to set *Experimental JavaScript flag* in order to try ES6 features. Firefox has an unstable version called *Firefox Nightly* that implements more features than the stable release of Firefox. If you are planning to write production JavaScript code using ES6 today, then targeting specific browsers is not a good option as your site will not reach to a large number of users.

To write JavaScript code using ES6, you can use some of the transpilers and pollyfills we mentioned earlier. Transpilers take ES6 code and convert it to its ES5 equivalent that today's browsers can understand. Traceur from Google happens to be the most popular transpiler of ES6. Other transpilers include 6to5, ES6-transpiler and a couple of others. The transpilers define certain commands using which we can convert ES6 code to ES5. We also have Grunt and Gulp packages to automate this process. Details about all these transpilers and their usage options are listed on Addy Osmani's ES6 tools page.

To check for a list of features supported by a particular browser or a transpiler, visit the ES6 compatibility table: http://kangax.github.io/compat-table/es6/

Setting up ES6 Environment

To run the code of this article, I have used traceur. To use traceur, you need to have Node.js and the following NPM package installed.

- npm install traceur -g

With this setup, you can compile any ES6 file using the following traceur command:

- traceur --out <destinationInES5>.js --script

<sourceInES6>.js

This command produces ES5 equivalent of the ES6 code. This can be referred on a page and loaded into a browser to test the behavior. Each ES6 source file in the sample code has the command to compile the file.

The ES5 files produced after traceur compilation contains traceur objects. We need the traceur library to be loaded before loading the compiled file. This library has to be installed using bower.

- bower install traceur --save-dev

Load the traceur library file and set the experimental flag to *true* before loading compiled scripts.

```
<script src="bower_components/traceur/
traceur.js"></script>
<script>
  traceur.options.experimental = true;
</script>
```

The step involving compilation of the source files can be automated using Grunt or, Gulp tasks. Addy Osmani's ES6 tools page lists these tasks. I am leaving it as an assignment for the reader to setup Grunt or Gulp.

Variables in ES6

The "let" keyword

The "var" variable specifier of JavaScript creates variables of two scopes: Global and Function. The var specifier cannot create block scoped variables. A variable declared at a block level is also hoisted at the function level. For example, consider the following snippet:

```
(function(){
  var numbers = [10, 20, 25, 90, 17, 38,
  61];
  var average=0;

  for(var count=0; count<numbers.length;
  count++){
    average += numbers[count];
  }

  average = average/numbers.length;
  console.log(average);
  console.log(count);
}());
```

The variable count is hoisted at the function level and so the last console.log() statement is valid. This behavior frustrates programmers that have worked on any other programming languages like C#, VB.NET or Java. There are good reasons for a variable to not exist beyond a block. This is particularly useful in case of for loops, where we don't need the counter to exist beyond the loop. ES6 introduced a new keyword, *let*, that makes it possible to create block level variables. In the function we just saw, replace *var* in the *for* loop with *let*:

```
(function(){
  var numbers = [10, 20, 25, 90, 17, 38,
  61];
  var average=0;

  for(let count=0; count<numbers.length;
  count++){
    average += numbers[count];
  }

  average = average/numbers.length;
  console.log(average);
  console.log(count);
}());
```

Now, the last statement would throw an error in strict mode, as the variable is not declared at the function level.

The "const" keyword

In the current version of JavaScript, there is no direct way to define a constant. In real applications, constants have significant importance. The only possible way I see is using *Object.freeze*, but this method is created for a different purpose. ES6 solves this problem with the *const* keyword.

The *const* keyword applies two restrictions on the variable:

- Prevents modification of value of the variable
- Sets block to the variable

In other words, one can also say that *const* is similar to *let* but with value modification prevented.

Following snippet demonstrates usage of *const*:

```
function getAreaOfCircle(radius) {
  if(radius) {
    const PI = 3.412;
    const R;  //Error, constant must be
    initialized
    PI=22/7;  //Error, PI cannot be
    reassigned
    return PI * radius * radius;
  }
  console.log(PI);  //Error, PI is block
  scoped return 0;
}

console.log(getAreaOfCircle(10));
console.log(getAreaOfCircle());
//console.log(PI);
```

Destructuring

The destructuring feature allows us to assign values to a set of variables from a stream of data without navigating through each entry in the stream. The stream can be an object, an array or any value of one of these types returned from a function. Consider the following object:

```
var topic = {name:'ECMAScript 6',
comment: 'Next version of JavaScript',
browserStatus: {
  chrome: 'partial',
  opera:'partial',
  ie: 'very less',
  ieTechPreview: 'partial'
}};
```

Let's assign values of name and comment to two different variables using destructuring:

```
var {name, comment} = topic;
```

The above statement declares the variables name and comment and assigns values of these properties in the object topic. If a property with identical name is not found in the object, the variable would contain *undefined*. It is also possible to destructure properties out of an object present at any level inside the main object. For example, the following snippet gets values assigned to *chrome* and *ieTechPreview* properties of the *browserStatus* object.

```
var {browserStatus:{chrome :
chromeSupport, ieTechPreview:
iePreviewSupport}} = topic;
```

Destructuring can be applied on an array of values as well. Following snippet shows this:

```
var numbers=[20, 30, 40, 50];

var [first, second, third ] = numbers;
console.log(first);      //20
console.log(second);     //30
console.log(third);      //40
```

It is possible to skip a few numbers out of the array and destructure rest of them. Following snippet shows this feature:

```
var [, , fourth, fifth ] = numbers;

console.log(fourth);     //40
console.log(fifth);      //50
```

Destructuring can also be applied on parameters passed to a function. The way to apply destructuring remains the same. Following is an example:

```
function add({firstNumber, secondNumber})
{
  return firstNumber + secondNumber;
}

console.log(add({firstNumber: 94,
secondNumber: 19}));
```

Spread

The *spread* operator is used to expand values in an array into individual elements. The spread list can be used to pass into a function as arguments or, as part of another array.

Syntax for using the *spread* operator is the array object prepended with three periods (dots). Following snippet shows the syntax and usage within a function:

```
function printPersonDetails(name, place,
occupation){
  console.log(name + " works at " +
  place + " as a/an " + occupation);
}
printPersonDetails(...['Ravi',
'Hyderabad', 'Software Professional']);
```

Here, the three values would be assigned to three parameters in the function. Spread operator can be used to make an array a part of another array as well.

```
var arr = [1, 2, 3];
var arr2 = [19, 27, 12,...arr, 63, 38];
console.log(arr2);
```

In the above snippet, the resulting array arr2 would contain 8 elements in it.

Functions in ES6

The Arrow function

As C# developers, we love lambdas. When we write code in another language, we miss this feature at times. ES6 has made us happy by including this feature in the JavaScript language.

When we write large scale JavaScript, callbacks and anonymous functions are inevitable. Most of the popular JavaScript libraries embrace extensive usage of callbacks and currying. Arrow functions add a syntactic sugar to the way we write anonymous functions and make development more productive.

For example, say we have a list of employees and we need to find the average of their age. We can do it using a callback passed into the *Array.forEach* function as shown below:

```
var employees = [{name:'Alex',age:33},
  {name:'Ben',age:29},
  {name:'Craig',age:57},
  {name:'Dan',age:27},
  {name:'Ed',age:48}];
var totalAge = 0;

employees.forEach(function(emp) {
  totalAge += emp.age;
});

console.log("Average age: " + totalAge /
employees.length);
```

Let's replace callback in the *forEach* function using an *arrow* function. By adding the arrow function, we can get rid of the function keyword and parentheses as we just have one argument. Following is the modified usage of *forEach* function:

```
employees.forEach(emp => {
  totalAge += emp.age;
});
```

If the callback has just one statement with a return value, we can get rid of the curly braces as well. To demonstrate this, let's find if every employee in the above array is aged over 30:

```
var allAbove30 = employees.every(emp =>
emp.age >= 30 );
console.log(allAbove30);
```

In addition to providing syntactic sugar, arrow functions ease usage of *this* reference in the callbacks. Callbacks are executed in the context of the object that calls them and so *this* reference inside a callback method refers to the object used to call the method. For example, consider the following snippet:

```
var events = {
  registeredEvents:[],
    register: function(name, handler){
      this.registeredEvents.push({name:
      name, handler: handler});
    },
    raise: function(name){
      var filtered = this.
      registeredEvents.filter( e =>
      e.name=== name)[0];
      filtered.handler();
    }
  };
  var worker1 = {
    name:'Ravi',
    registerWorkDone: function(){
    events.register('workDone',
    function(){
      console.log(this.name + "'s
      current task is done! Free for new
      assignment.");
    });
  }
};
worker1.registerWorkDone();
events.raise('workDone');

//Output: workDone's current task is
done! Free for new assignment.
```

Here, we have an event object and a worker object. The worker object registers an event, in which it attempts to print name of the worker. But as the event is called by the event entry object, where name refers to name of the event, it prints workDone instead of printing Ravi. In order to avoid it, we need to cache the *this* reference in a local variable and use it. Following snippet shows this:

```
var worker1 = {
  name:'Ravi',
  registerWorkDone: function(){
    var self=this;
    events.register('workDone',
    function(){
      console.log(self.name + "'s
      current task is done! Free for new
      assignment.");
    });
  }
};

//Output: Ravi's current task is done!
Free for new assignment.
```

If we use *arrow* functions, we don't need to deal with these difficulties. Arrow functions are executed in the context of the containing object and so *this* reference would always refer to the object inside which the function is written.

```
var worker2 = {
  name:'Kiran',
  registerWorkDone: function(){
    events.register('workDone2', () => {
      console.log(this.name + "'s
      current task is done! Free for new
      assignment.");
    });
  }
};
worker2.registerWorkDone();
events.raise('workDone2');

//Output: Ravi's current task is done!
Free for new assignment.
```

Default Parameters

Another feature that is commonly available in other programming languages is providing default values to parameters of functions. Value of the parameter would be set to default if the value is not passed while calling the function. Today we can do it in JavaScript, by checking if the parameter has any value.

```
function sayHi(name, place){
  name = name || 'Ravi';
  place = place || 'Hyderabad';
  console.log("Hello, " + name +" from "
  + place + "!");
}
```

We need to add a statement for every parameter that needs a default value and there are good possibilities of missing the parameters. ES6 adds the feature of specifying default values to the parameters *inline* in the function signature. Default value is assigned when either value is not passed in for the parameter or, set to null. Following snippet demonstrates some use cases of default values:

```
function sayHi(name = 'Ravi', place =
'Hyderabad'){
    console.log("Hello, " + name +" from
" + place + "!");
}
sayHi();
sayHi('Ram');
sayHi('Hari','Bengaluru');
sayHi(undefined, 'Pune');
```

Value of one parameter can be used to calculate default value of another parameter. Let's add another parameter message to the function above. It would use the parameter name in its value:

```
function sayHi(name = 'Ravi', place =
'Hyderabad', message = "No additional
message for " + name){
    console.log("Hello, " + name +" from
" + place + "! (" + message + ")");
}
```

```
sayHi('Ram');
sayHi('Hari','Bengaluru', 'Message from
Rani, Bengaluru');
```

Rest Parameters

The *Rest* parameter allows the caller to specify any number of arguments to the function. All passed values are stored in a single array (similar to *params* in C#).

Now you may think what's the difference between a rest parameter and arguments? Basic difference is, arguments is not an array. It is an object with some properties taken from array; whereas a rest parameter is an array.

Syntax to specify Rest parameter is by prefixing the parameter with three periods (...). Following snippet shows the syntax:

```
function addNumbers(...numbers){
  var total = 0;
  numbers.forEach(number => {
    total += number;
  });
  console.log(total);
}
```

You can call this function by passing any number of values (numbers) to it and it would print sum of all the numbers. Other parameters can also be accepted along with *Rest* parameters, but it is advised to always use *Rest* parameter as the last parameter.

```
function
addEmployeesToDepartment(departmentId,
...employees){
  var department=getDepartment();
  employees.forEach(emp => {
    department.employees.push(emp);
  });
}
```

Classes

Constructor, Properties, Methods

While writing large JavaScript applications, we may want to enclose certain pieces of logic inside units like *classes*. Classes could be nice abstraction layers in JavaScript apps as they can contain data and logic together. Though we cannot write classes in JavaScript like we do in Object Oriented languages, we can write similar constructs using *prototype* property of objects. We also can inherit new classes using the *prototype* property.

There is a learning curve around the *prototype* property before one can use it comfortably to write OOPs like code in JavaScript. To ease this, ES6 includes classes. Using ES6 class feature, we can write classes, create objects, inherit and override features of parent class inside child classes. Following snippet shows syntax of writing a class:

```
class City{

  constructor(name, country){
    this._name=name;
    this._country=country;
  }

  startDay(){
    return "Good Morning!";
  }

  get name(){
    return this._name;
  }

  set name(value){
    this._name=value;
  }

  get country(){
    return this._country;
  }

  set country(value){
    this._country=value;
  }
}
```

As you can see, this class has the following components:

- A constructor: The constructor is defined using the *constructor* keyword. Here, the constructor accepts two parameters and sets them to class members

- Two properties, name and country: Keywords for defining properties are get and set. You can create a property with only *get*; that will be a read only property. Properties in the city class are used to get value of and set value to the class members _name and _country respectively. Names of property and its corresponding member should be different; otherwise it would lead to an infinite recursion.

- A method, *startDay*: The syntax of the method may have puzzled you, as it doesn't have the function keyword attached. This is the new way of creating methods in ES6. This syntax can be used to define methods inside objects as well.

```
var city= {
  startDay()
  {
    return "Starting day...";
  }
};
```

Now that we have defined a class and added some members to it, we can create an object and play with it. Syntax for creating an instance of the class is also similar to the one we do in OOPs based languages.

```
let city = new City("Hyderabad",
"India");
console.log(city.startDay());
console.log(city.name);
city.name="Bengaluru";
console.log(city.name);
```

Inheritance

Classes can be inherited from other classes using *extends* keyword. All members of parent class are accessible using objects of child class. Members of parent class can be invoked from child class using *super* keyword.

Let's write a child class, *MetroCity* by deriving from *City* class. The first thing that we would like to do while writing a child class is to invoke constructor of the parent class. Parent class constructor can be called using *super* keyword. Calling of parent class constructor need not be the first statement in the child class; but it is a good practice to do it as the first thing.

```
class MetroCity extends City
{
  constructor(name, country,
  hasMetroRail)
  {
    super(name, country);
    this._hasMetroRail = hasMetroRail;
  }
}
```

Let's override the *startDay* method inside the child class. Following is the snippet:

```
startDay(){
  return super.startDay() + "\n Good
Morning from Radio 365!";
}
```

Notice the way it uses the keyword *super* to call the parent class method. Now that the child class is ready, let's create an object and call the *startDay* method.

```
let metroCity = new MetroCity("New
Delhi", "India", true);
console.log(metroCity.startDay());
```

This snippet prints the combined message from parent class and child class.

Iterators in ES6

Built-in Iterators

ES6 has iterators to help us loop through arrays. Iterators are objects with a method called *next* implemented on it and this method returns an object that has the following properties:

- value: Value of the occurrence
- done: A Boolean value indicating if the last element is reached

The simplest to understand iterators is using one of the built-in iterators. Every array in ES6 has an iterator associated with it. It can be invoked using *values()* method on any array. Following snippet

shows usage of this iterator using a while loop:

```
let sum = 0;
let numbers = [10, 20, 30, 40, 50];
let iterator = numbers.values();
let nextNumber = iterator.next();
while(!nextNumber.done){
  sum += nextNumber.value;
  nextNumber = iterator.next();
}
```

The *next* method helps in iterating through the array and fetching the next item from the array with status of iteration. The *while* loop continues till the last element in the array. Last element returned by the iterator has *value* set to null and *done* set to true.

For...of loop

A new loop has been added in ES6 to ease iteration through iterators. Using the for...of loop, we can iterate without calling the *next* method and checking *done* property explicitly. The array internally performs all the required logic to continue iteration. Following snippet shows the syntax of using for...of loop.

```
for(let n of numbers){
  sum += n;
}
```

This loop does the same work as the while loop we just saw.

Defining Custom Iterators

We can define custom iterators to iterate over object of a class. Say, we have a shelf class with an array containing books in it.

```
class Shelf{
  constructor(){
    this.books = [];
  }

  addBooks(...bookNames){
    bookNames.forEach(name => {
      this.books.push(name);
    });
  }
}
```

Let's add an iterator to make objects of this class iterable. In C#, we need to implement the *IEnumerable* interface on a class to make it iterable. In ES6, we don't have interfaces, but we need to implement a special type of function to make the objects iterable.

ES6 has a special object, *Symbol*. Value of the *Symbol* object is unique on every usage. This object contains a set of well-known symbols, including *iterator*. The iterator returns default *iterator* for an object and it can be used to create an iterable class.

To make the above Shelf class iterable, we need to add a method to the class and name it as *iterator* symbol. Presence of this method indicates that an object of this class is iterable. The method should do the following:

- return an object containing the method *next*; this method is used to continue iteration.

- the *next* method should return an object containing value of the entry and a Boolean property *done* to indicate if the iteration is completed.

Following is the iterator method:

```
[Symbol.iterator](){
  let obj=this;
  let index = 0;
  return {
    next: function(){
      var result = {value: undefined,
      done: true};
      if(index < obj.books.length){
        result.value = obj.books[index];
        result.done=false;
        index += 1;
      }
      return result;
    }
  }
}
```

Now, you can create an object of the *Shelf* class and iterate through it using for...of loop.


```
var shelf = new Shelf();
shelf.addBooks('Programming ASP.
NET MVC 5','Getting lost in
JavaScript','AngularJS in Action','C# 6:
What\'s new in the language' );
for(let book of shelf){
  console.log(book);
}
```

Generators

These techniques of creating iterators that we just saw may not always be the best option as we need to follow a specified set of rules to make a class iterable.

Generators make the process of creating iterators easier. Generators follow a special syntax. Following are some features of the syntax:

- A generator function has an asterisk in its name
- Uses *yield* keyword to return values

Use of the *yield* keyword reduces a lot of work for us. A function can return multiple values using the *yield* keyword and it is smart enough to remember state of the last iteration and return next value when asked in the subsequent call.

Let's rewrite the above iteration function using *yield* keyword.

```
*[Symbol.iterator](){
    for(let i=0;i<this.books.length;
    i++){
      yield this.books[i];
  }
}
```

This would produce the same result as the function we wrote earlier; but in this case we got the result with less code.

Conclusion

ES6 comes with a number of features that makes JavaScript a better language to work with in large scale applications and adds a lot of useful syntactic sugar at the same time. Many companies (including AngularJS team) have already started using ES6 on a daily basis. Being web developers, we cannot keep waiting for the final call to learn new features of our favorite programming language. This article covered features of ES6 that makes JavaScript look and feel better; we will explore the new APIs and some more features of ES6 in forthcoming articles. Stay tuned ■

Download the entire source code from our GitHub Repository at bit.ly/dncm16-ecmascript

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

AGILE IS NOT FOR THE FAINT OF HEART

SOIL NUTRIENTS

This is not the column I had planned to write. But in October I went to Silicon Valley Code Camp where I attended an excellent session called "Agile in a Waterfall World" by Jeanne Bradford. Her company, TCGen, did research into using Agile in hardware development and manufacturing. Now, this column is about software, not hardware; but much of that research and what she said in her session is applicable to software gardeners.

So let's go back to one of my first columns Software Gardening: Using the Right Soil where I discussed soil. As any good gardener will tell you, soil needs the right nutrients and different types of plants need different nutrients. It's also important the soil not have things that are bad for the plants. You need to know about the soil you already have. That sometimes requires a detailed analysis of the soil. In my earlier column I also said that as software gardeners, our soil is *agile practices*.

What I decided to write about in this issue is to analyze Agile to see what it really means. Jeanne has graciously allowed me to use parts of her presentation in this column. If you want more, you can get her entire slide deck at http://www.tcgen.com/tools.

We'll start by going back to the Agile Manifesto. On that winter day in 2001 at Snowbird, a ski resort just outside of my city of Salt Lake City, Utah, the 17 original signers of the Manifesto came up with four main points (see Figure 1).

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

Figure 1: The main points of the Agile Manifesto

But there is more to it than those four points. There are actually 12 principles that support the four main ones. You can read them at http://www.agilemanifesto.org/principles.html. In Figure 2, I've taken Jeanne's rewording and reordering of those 12 principles. Take a moment to read that reordering. I'll be here when you get back.

1. Business people and developers work together daily
2. Projects require motivated individuals, support & trust
3. Face-to-face conversation is most efficient
4. Agile processes promote sustainable development
5. Continuous attention to technical excellence
6. Simplicity – is essential
7. The best designs emerge from self-organizing teams
8. At regular intervals, the team reflects
9. Welcome changing requirements
10. Continuous delivery of valuable software
11. Deliver working software frequently
12. Working software is the measure of progress

Figure 2: The 12 supporting principles of the Agile Manifesto

Did you see what's important in that reordering? No? Read the first nine again. None of them have anything to do specifically with software. That's nine of 12 or 75%. In other words, the success of your project is 75% based on **non-technical** skills and only 25% on technical skills. The non-technical principles can be applied anywhere and on any team. That leaves only the last three as software specific.

This is really important. It means that nine of the top 12 nutrients for your soil are not targeting software development, but rather targeting the general health of your team. It includes developers, QA, devops, management, business owners… everyone involved. It means that you need excellent interpersonal (how do you work together) and soft skills (the non-technical skills). It means if you want your project to succeed, you need to concentrate more on these types of skills rather than technical skills. This is not easy. In fact, I'll bet it's easier to cultivate the technical skills over the non-technical skills.

These non-technical skills include things like presentations, public speaking, understanding the business domain, willingness to get the job done, ability to talk face-to-face, ability to explain extremely technical concepts to a non-technical person, trusting team members, interacting with them daily, etc. Can you do these things? If not, you should concentrate your learning on them. How about your team members? Can they do these things?

In their research TCGen asked their customers "What are the most impactful elements of Agile/Scrum applied to software?" The four things that stood out as having the biggest impact were daily standups, burn down charts, team culture, and customer owner. Are you doing daily standups? Do you have burn down charts? Is your customer owner engaged in the project?

So why is applying Agile so hard? This is another question Jeanne addressed. The simple answer is Agile is hard. It requires higher process literacy and greater cross-functional teamwork skills. Your team must be sophisticated. I once worked in a company where water-fall development was the way things had always been done. The Vice-President of Development wanted to adopt Scrum. She read a couple of books on the subject then had development and project managers read the same books. When Scrum was rolled out to the organization, it failed. Teams were convinced that Agile was worthless.

The proper way to do this would have been to have key players attend training classes, then hire an experienced Scrum Master and/or Coach and roll it out to one team then move on to the next team. The bottom line is ***Agile is not for the faint of heart.***

To be successful with Agile, your team must first adopt some cultural aspects:

- Teams must have high performance

- Teams must be self-organized
- Teams must have trust and empowerment
- The customer owner and team leader must interact daily
- Teams must have daily standup meetings
- Teams must accept the fact that requirements can (and probably will) change

Next, create boundary conditions. These are the conditions that you will use to help identify what goes into a sprint. The top of the list is a User Story, which is a short explanation of how a user will use a particular piece of functionality. These user stories are the Product Attributes. Then add Program Attributes or budget and schedule.

When you have these, pick the top three to seven. You can now further define your boundary conditions. In fact, there are five primary conditions to consider - features, product cost, development cost, schedule, and quality (see Figure 3). If you can keep the team progressing inside these conditions, you will have success.

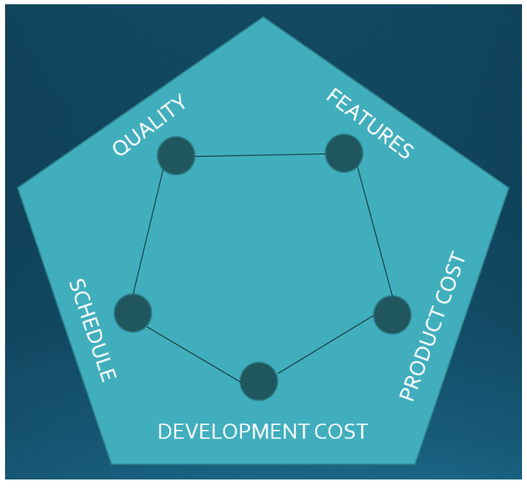


Figure 3: Boundary conditions

As long as you stay inside the boundary conditions, everything runs fine in the sprint and the project. But what happens when something doesn't work as planned? Say, you lose a key team member (Figure 4). The schedule could slip and suddenly you're outside the boundary. If you are continuously measuring all aspects of your project, you'll see this issue and adjust accordingly.

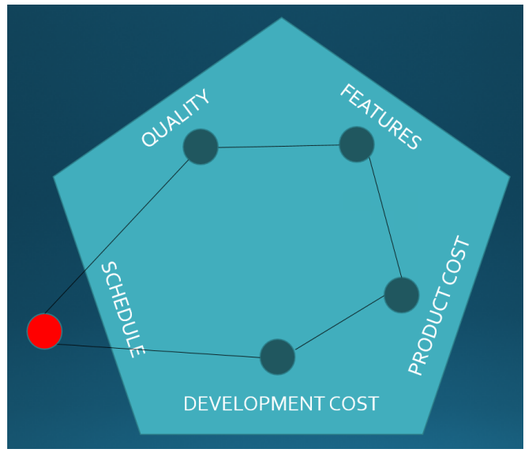


Figure 4: Out of bounds condition

Perhaps a schedule change will be sufficient. Or you may need to change other aspects of your project, maybe dropping features and dealing with cost adjustments. I strongly encourage you to not decrease quality. This adds technical debt and actually will increase costs long term.

Now you can move to step 2, attacking your burn down. For each sprint, you need to identify tasks you can get done. Track how many you accomplish over time. In the beginning of your project, you'll probably do more work on definitions and infrastructure. The middle is mostly task-based work, getting working code out the door. Later in the project schedule, you'll probably do more validation.

But what happens if you get to the end of sprint and you haven't finished all the tasks planned for the sprint? You have two choices. First, you can extend the length of the sprint to finish them or second, you can schedule to finish the work on another sprint. Pick one, but pick correctly. Did you pick option two? If so, good for you. Don't slip the sprint. As you gain more experience with Agile and better understanding of the business domain, you'll be able to more accurately split up stories into tasks and estimate how long each task will take.

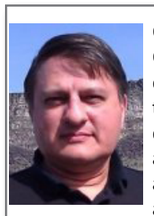
Finally, do not skip the retrospective. Key learning goes on here. It's where you and the entire team, including the business owners, discuss what went right and wrong and how you can fix it. Do a retrospective often, perhaps a short one at the end of each sprint and a longer, more in-depth retrospective at the end of the project.

You should now be able to look at the nutrients in your soil. How well is Agile working for you? Where do you need to make changes? How can you make changes so that Agile works for you? By having the proper nutrients in your soil you can ensure that your software garden is lush, green, and growing.

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.

Moving forward with .NET

Design Patterns

Design Patterns.... yes, there are several books and resources written on this topic. When it comes to Software Development, Design Patterns promotes constancy across the code base and allows us to develop better maintainable software.

There are many Design Patterns in Software Development. Some of these patterns are very popular. It is almost true to say that most patterns can be embraced irrespective of the programming language we choose.

Based on the type of application, we may use one or more Design Patterns. Sometimes we may mix and match Design Patterns as well. It is very important to consider that we primarily want to use these patterns as *effective communication* tools. This may not be obvious at this stage. However during this article we will look at how the Design Patterns are utilised in various applications.

In this article we will not just focus on a set of Design Patterns. We well take a fresh view of some of the existing Design Patterns and see how we can go about using them in real world dilemmas and concerns.

Bit of a background

It is a fact that some developers hate Design Patterns. This mainly because of analysing, agreeing and implementing a particular Design Pattern can be a headache. I'm pretty sure we have all come across situations where developers spend countless hours, if not days, discussing the type of pattern to use, the best approach and the way to go about implementing it. This is a very poor way to develop software.

This dilemma is often caused by thinking that their code can fit into a set of Design Patterns. This is an incredibly hard thing to do. But if we think Design Patterns are a set of tools which allows us to make good decisions and can be used as an effective communication tool; then we are approaching it in the right way.

This article mainly focuses on .NET Design Patterns using C# as the programming language. Some Patterns may be applied to non .NET based programming languages as well. Repeating what I said earlier, most patterns can be embraced irrespective of the programming language we choose.

Abstract Factory Pattern

Wikipedia definition:

> The **abstract factory pattern** provides a way to encapsulate a group of individual **factories** that have a common theme without specifying their concrete classes.

While this definition is true, the real usage of this pattern can be varying. It can be based on real life concerns and problems people may have. In its simplest form, we would create instances with related objects without having to specify their concrete implementations. Please refer to the following example:

```
public class Book
{
  public string Title { get; set; }
  public int Pages { get; set; }

  public override string ToString()
  {
    return string.Format("Book {0} -
    {1}", Title, Pages);
  }
}

public class Program
{
  public static void Main(string[] args)
  {
    Console.WriteLine(CreateInstance
    ("ConsoleApplication1.Book",
    new Dictionary<string, object>()
    {
      {"Title", "Gulliver's Travels"},
      {"Pages", 10},
    }));
  }
```

```
  private static object
  CreateInstance(string className,
  Dictionary<string, object> values)
  {
    Type type = Type.
    GetType(className);
    object instance = Activator.
    CreateInstance(type);

    foreach (var entry in values)
    {
      type.GetProperty(entry.Key).
      SetValue(instance, entry.
      Value, null);
    }

    return instance;
  }
}
```

As per the above example, creation of instances are delegated to a routine called *CreateInstances()*. This routine takes a class name and the property values as arguments.

At the first glance, this seems like a lot of code just to create an instance and add some values to its properties. But the approach becomes very powerful when we want to dynamically create instances based on parameters. For example, creating instances at runtime based on User Inputs. This is also very centric to Dependency Injection (DI). The above example just demoes the fundamental of this pattern. But the best way to demonstrate Abstract Factory pattern is to take a look at some real world examples. It would be redundant to introduce something already out there. Therefore if you are interested, please see this Stack Overflow question, which has some great information.

Additional Note: *Activator.CreateInstance* is not centric to Abstract Factory Pattern. It just allows us to create instances in a convenient way based on the type parameter. In some cases we would just create instances by new'ing up (i.e new Book()) and still use the Abstract Factory Pattern. It all depends on the use case and their various applications.

Cascade Pattern

I'm sure we often see code patterns like the following:

```
public class MailManager
{
  public void To(string address) {
  Console.WriteLine("To");}
  public void From(string address) {
  Console.WriteLine("From"); }
  public void Subject(string subject) {
  Console.WriteLine("Subject"); }
  public void Body(string body) {
  Console.WriteLine("Body"); }
  public void Send() { Console.
  WriteLine("Sent!"); }
}

public class Program
{
  public static void Main(string[] args)
  {
    var mailManager = new MailManager();
    mailManager.From("alan@developer.
    com");
    mailManager.To("jonsmith@developer.
    com");
    mailManager.Subject("Code sample");
    mailManager.Body("This is an the
    email body!");
    mailManager.Send();
  }
}
```

This is a pretty trivial code sample. But let's concentrate on the client of the MailManager class, which is the class Program. If we look at this class, it creates an instance of MailManager and invokes routines such as .To(), .From(), .Body() .Send() etc.

If we take a good look at the code, there are a couple of issues in writing code like we just saw.

a. Notice the variable "*mailManager*". It has been repeated number of times. So we feel somewhat awkward writing redundant writing code.

b. What if there is another mail we want to send out? Should we create a new instance of MailManager or should we reuse the existing "*mailManager*" instance? The reason we have these questions in the first place is that the API (Application Programming Interface) is not clear to the consumer.

Let's look at a better way to represent this code.

First, we make a small change to the MailManager class as shown here. We modify the code so we could return the current instance of the MailManager instead of the return type *void*.

Notce that the Send() method does not return the MailManager. I will explain why we did this is in the next section.

Modified code is shown here.

```
public class Mailmanager
{
  public MailManager To(string address)
  { Console.WriteLine("To"); return
  this; }
  public MailManager From(string
  address) { Console.WriteLine("From");
  return this; }
  public MailManager Subject(string
  subject) { Console.WriteLine("Subject"
  ); return this;}
  public MailManager Body(string body) {
  Console.WriteLine("Body"); return
  this; }
  public void Send() { Console.
  WriteLine("Sent!"); }
}
```

In order to consume the new MailManager implementation, we will modify the Program as below.

```
public static void Main(string[] args)
{
  new MailManager()
    .From("alan@developer.com")
    .To("jonsmith@developer.com")
    .Subject("Code sample")
    .Body("This is an the email body!")
    .Send();
}
```

The duplication and the verbosity of the code have been removed. We have also introduced a nice fluent style API. We refer to this as the Cascade pattern. You probably have seen this pattern in many popular frameworks such as FluentValidation. One of my favourites is the NBuilder.

```
Builder<Product>.CreateNew().With(x =>
x.Title = "some title").Build();
```

Cascade-Lambda pattern

This is where we start to add some flavour to the Cascade Pattern. Let's extend this example a bit more. Based on the previous example, here is the code we ended up writing.

```
new MailManager()
  .From("alan@developer.com")
  .To("jonsmith@developer.com")
  .Subject("Code sample")
  .Body("This is an the email body!")
  .Send();
```

Notice that the Send() method is invoked from an instance of the MailManager. It is the last routine of methods chain. Therefore it does not require returning an instance. This also means the API implicitly indicates that if we want send another mail, we will have to create a new MailManager instance. However it is not explicitly clear to the user what we should do after the call to .Send().

This is where we can take the advantage of lambda expressions and make the intention explicit to the consumer of this API.

First we convert the Send() method to a Static method and change its signature to accept an Action delegate. This delegate takes MailManager as a parameter. We invoke this action within the Send() method as shown here:

```
public class MailManager
{
  public MailManager To(string address)
  { Console.WriteLine("To"); return
  this; }
  public MailManager From(string
  address) { Console.WriteLine("From");
  return this; }
  public MailManager Subject(string
  subject) { Console.WriteLine("Subject
  "); return this;
}
  public MailManager Body(string body) {
  Console.WriteLine("Body"); return
  this; }

  public static void
  Send(Action<MailManager> action)
  {
    action(new MailManager());
```

```
    Console.WriteLine("Sent!");
  }
}
```

In order to consume the MailManager class, we can change the Program as seen here:

```
Mailmanager.Send((mail) => mail
  .From("alan@developer.com")
  .To("jonsmith@developer.com")
  .Subject("Code sample")
  .Body("This is an the email body!"));
```

As we see in the code sample, the action specified by the delegate as an argument to the Send() method clearly indicates that the action is related to constructing a *mail*. Hence it can be sent out by calling the Send() method. This approach is much more elegant as it removes the confusions around the Send() method, which I described earlier.

Pluggable pattern

The best way to describe the pluggable behaviour is to use an example. The following code sample calculates the total of a given array of numbers.

```
public class Program
{
  public static void Main(string[] args)
  {
    Console.WriteLine(GetTotal(new []
    {1, 2, 3, 4, 5, 6}));
    Console.Read();
  }

  public static int GetTotal(int[]
  numbers)
  {
    int total = 0;

    foreach (int n in numbers)
    {
      total += n;
    }

    return total;
  }
}
```

Let's say we have a new requirement. Although we do not want to change the GetTotal() method, but we would also like to calculate only even numbers. Most of us would add another method, say GetEvenTotalNumbers as shown here.

```
public class Program
{
  public static int GetTotal(int[]
  numbers)
  {
    int total = 0;

    foreach (int n in numbers)
    {
      total += n;
    }

    return total;
  }

  public static int
  GetTotalEvenNumbers(int[] numbers)
  {
    int total = 0;

    foreach (int n in numbers)
    {
      if (n%2 == 0)
      {
        total += n;
      }
    }

    return total;
  }

  public static void Main(string[] args)
  {
    Console.WriteLine(GetTotal(new []
    {1, 2, 3, 4, 5, 6}));
    Console.WriteLine
    (GetTotalEvenNumbers(new[] { 1, 2, 3,
    4, 5, 6 }));
    Console.Read();
  }
}
```

We just copied/pasted the existing function and added the only condition that requires calculating even numbers. How easy is that! Assuming there is another requirement to calculate the total of odd numbers, it is again as simple as copying/pasting one of the earlier methods, and modifying it slightly to calculate odd numbers.

```
public static int
GetTotalOddNumbers(int[] numbers)
{
  int total = 0;

  foreach (int n in numbers)
  {
    if (n % 2 != 0)
    {
      total += n;
    }
  }

  return total;
}
```

At this stage we probably realize that this is not the approach we should take to write software. It is pretty much copy paste unmaintainable code. Why is it unmaintainable? Let's say if we have to make a change to the way we calculate the total. This means we would have to make changes in 3 different methods.

If we carefully analyse all 3 methods, they are very similar in terms of their implementation. Only difference is the *if* condition.

In order to remove the code duplication we can introduce the Pluggable Behaviour.
We can externalize the difference and inject it into a method. This way the consumer of the API has control over what has been passed into the method. This is called the Pluggable Behaviour.

```
public class Program
{
  public static int GetTotal(int[]
  numbers, Predicate<int> selector)
  {
    int total = 0;
    foreach (int n in numbers)
    {
      if (selector(n))
      {
        total += n;
      }
    }

    return total;
  }

  public static void Main(string[] args)
  {
    Console.WriteLine(GetTotal(new []
    {1, 2, 3, 4, 5, 6}, i => true));
    Console.WriteLine(GetTotal(new[] {
    1, 2, 3, 4, 5, 6 }, i => i % 2 ==
    0));
    Console.WriteLine(GetTotal(new[] {
    1, 2, 3, 4, 5, 6 }, i => i % 2 !=
    0));
    Console.Read();
  }
}
```

As we see in the above example, a Predictate<T> has been injected to the method. This allows us to externalize the selection criteria. The code duplication has been removed, and we have much more maintainable code.

In addition to this, let's say we were to extend the behaviour of the selector. For instance, the selection is based on multiple parameters. For this, we can utilize a *Func* delegate. You can specify multiple parameters to the selector and return the result you desire. For more information on how to use *Func* delegate please refer to Func<T1, T2, T3, TResult>.

Execute Around Pattern with Lambda Expressions

This pattern allows us to execute a block of code using lambda expression. Now that sounds very simple and that's what lambda expressions do. However this pattern is about using lambda expressions and implements a coding style, which will enhance one of the existing popular Design Patterns. Let's see an example.

Let's say we want to clean-up resources in an object. We would write code similar to the following:

```
public class Database
{
  public Database()
  {
    Debug.WriteLine("Database
    Created..");
  }

  public void Query1()
  {
    Debug.WriteLine("Query1..");
  }

  public void Query2()
  {
    Debug.WriteLine("Query2..");
  }
```

```
  ~Database()
  {
    Debug.WriteLine(“Cleaned-Up”);
  }
}

  public class Program
  {
    public static void Main(string[]
    args)
    {
      var db = new Database();
      db.Query1();
      db.Query2();
    }
  }
```

The output of this program would be..

```
Database Created..
Query1..
Query2..
Cleaned Up!
```

Note that the Finalizer/Destructor implicitly gets invoked and it will clean-up the resources. The problem with the above code is that we don't have control over ***when*** the Finalizer gets invoked.

Let's see the same code in a for loop and execute it a couple of times:

```
public class Program
{
  public static void Main(string[] args)
  {
    for (int i = 0; i < 4; i++)
    {
      var db = new Database();
      db.Query1();
      db.Query2();
    }
  }
}
```

The Program would produce the following output.

```
Database Created..
Query1..
Query2..
Database Created..
Query1..
Query2..
Database Created..
Query1..
Query2..
Database Created..
Query1..
Query2..
Cleaned Up!
Cleaned Up!
Cleaned Up!
Cleaned Up!
```

All clean up operations for each DB creation happened at the end of the loop!

This may not be ideal if we want to release the resources explicitly so they don't live in the managed heap too long before being Garbage Collected. In a real world example, there can be so many objects having a large object graph, trying to create database connections and timing out. The obvious solution is to clean-up the resources explicitly and as quickly as possible.

Let's introduce a Cleanup() method as seen here:

```
public class Database
{
  public Database()
  {
    Debug.WriteLine(“Database
    Created..”);
  }

  public void Query1()
  {
    Debug.WriteLine(“Query1..”);
  }

  public void Query2()
  {
    Debug.WriteLine(“Query2..”);
  }

  public void Cleanup()
  {
    Debug.WriteLine(“Cleaned Up!”);
  }

  ~Database()
  {
    Debug.WriteLine(“Cleaned Up!”);
  }
}

public class Program
{
  public static void Main(string[] args)
  {
    for (int i = 0; i < 4; i++)
    {
      var db = new Database();
      db.Query1();
      db.Query2();
      db.Cleanup();
    }
  }
}
```

```
Database Created..
Query1..
Query2..
Cleaned Up!
Database Created..
Query1..
Query2..
Cleaned Up!
Database Created..
Query1..
Query2..
Cleaned Up!
Database Created..
Query1..
Query2..
Cleaned Up!
Cleaned Up!
Cleaned Up!
Cleaned Up!
Cleaned Up!
```

Note that we have not removed the Finalizer yet. For each database creation Cleanup() will perform explicitly. As we saw in the first for loop example, at the end of the loop, the resources will be garbage collected.

One of the problems with this approach is that if there is an exception in one of the Query operations, the clean-up operation would never get called.

As many of us do we wrap the query operations in a try{} block and a finally {} block and perform the clean-up operation. Additionally we catch{} the exception and do something, but I have ignored that for code brevity.

```
public class Program
{
  public static void Main(string[] args)
  {
    for (int i = 0; i < 4; i++)
    {
      var db = new Database();
      try
      {
        db.Query1();
        db.Query2();
      }
      finally
      {
        db.Cleanup();
      }
    }
  }
}
```

Technically this solves the problem. As the clean-up operation always gets invoked regardless of whether there is an exception or not.

However this approach still has some other issues. For instance, each and every time when we instantiate the Db and invoke query operations, as developers we have to remember to include it in the *try{}* and f*inally{}* blocks. To make things worse, in more complex situations, we can even introduce bugs without knowing which operation to call etc.

So how do we tackle this situation?

This is where most of us would use the well-known Dispose Pattern. With the Dispose Pattern *try{}* and *finally{}* are no longer required. The *IDisposable. Dispose()* method cleans-up the resources at the end of the operations. This includes any exception scenarios during query operations.

```
public class Database : IDisposable
{
//More code..
  public void Dispose()
  {
    Cleanup();
    GC.SuppressFinalize(this);
  }
}

public class Program
{
  public static void Main(string[] args)
  {
    for (int i = 0; i < 4; i++)
    {
      using (var db = new Database())
      {
        db.Query1();
        db.Query2();
      }
```

```
  }
}
```

This is definitely a much better way to write the code. The *using* block abstracts away the dispose of the object. It guarantees that the clean-up will occur *using* Dispose() routine. Most of us would settle with this approach. You will see this pattern used in many applications.

But if we really look closely there is still a problem with the *using* pattern itself. Technically it will do the right thing by explicitly cleaning-up resources. But there is no guarantee that the client of the API would use the using block to clean-up-up the resources. For example, anyone can still write the following code:

```
var db = new Database();
db.Query1();
db.Query2();
```

For resource intensive applications, if this code has been committed without being noticed, this can have an adverse effect on the application. So we are back to square one. As we have noticed, there is no immediate dispose or clean-up operation that takes place.

A chance of missing the *Dispose()* method is a serious problem. Not to mention we are also presented with a new challenge of making sure we implement the Dispose method/logic correctly. Not everyone knows how to implement the Dispose method/logic correctly. Most would resort for some other resources such as blogs/articles. This is all unnecessary trouble to go through.

So in order to address these issues, it would be ideal if we can change the API in such a way that developers cannot make mistakes.

This is where we can use Lambda Expressions to resolve these issues. In order to implement the Execute Around Pattern with Lambda Expressions we will modify the code with the following:

```
public class Database
{
  private Database()
  {
    Debug.WriteLine("Database
    Created..");
  }

  public void Query1()
  {
    Debug.WriteLine("Query1..");
  }

  public void Query2()
  {
    Debug.WriteLine("Query2..");
  }

  private void Cleanup()
  {
    Debug.WriteLine("Cleaned Up!");
  }

  public static void
  Create(Action<Database> execution)
  {
    var db = new Database();

    try
    {
      execution(db);
    }
    finally
    {
      db.Cleanup();
    }
  }
}
```

There are few interesting things happening here. IDisposable implementation has been removed. The constructor of this class becomes private. So the design has been enforced in such a way that the user cannot directly instantiate the Database instance. Similarly the *Cleanup()* method is also private. There is the new Create() method, which takes an Action delegate (which accepts an instance of the database) as a parameter. The implementation of this method would execute the action specified by the Action<T> parameter. Importantly, the execution of the action has been wrapped in a *try{} finally{}* block, allowing to clean-up operation as we saw earlier.

Here is how the client/user consumes this API:

```
public class Program
{
  public static void Main(string[] args)
  {
    Database.Create(database =>
    {
      database.Query1();
      database.Query2();
    });
  }
}
```

The main difference from the previous approach is that now we are abstracting the clean-up operation from the client/user and instead are guiding the user to use a specific API. This approach becomes very natural as all boilerplate code has been abstracted away from the client. This way it is hard to imagine that the developer would make a mistake.

More Real World Applications of Execute Around Method Pattern with Lambda Expression

Obviously this pattern is not limited to managing resources of a database. It has so many other potentials. Here are some of its applications.

- In Transactional code where we create a transaction and check whether the transaction is completed, then commit or rollback when required.

- If we have heavy external resources that we want to dispose as quickly as possible without having to wait for the .NET Garbage collection.

- To get around with some framework limitations – more on this shortly. This is quite interesting. Please see the following:

Tackling Framework Limitations

I'm sure most of you are familiar with Unit Testing. I'm a huge fan of Unit Testing myself. In .NET platform, if you have used MSTest framework, I'm sure you have seen the ExpectedException attribute. There is a limitation on the usage of this attribute where we cannot specify the exact call that throws an exception during the test execution.

For example, see the test here.

```
[TestClass]
public class UnitTest1
{
  [TestMethod]
  [ExpectedException(typeof(Exception))]
  public void
  SomeTestMethodThrowsException()
  {
    var sut = new SystemUnderTest("some
    param");

    sut.SomeMethod();
  }
}
```

The code demoes a typical implementation of an ExpectedException attribute. Note that we expect sut.SomeMethod() would throw an exception. Here is how the SUT (System Under Test) would look like. Note that I have removed the detailed implementation for code brevity.

```
public class SystemUnderTest
{
  public SystemUnderTest(string param)
  {
  }

  public void SomeMethod()
  {
    //more code
    //throws exception
  }
}
```

During test execution, if there is an exception being thrown, it will be caught and the test would succeed. However the test would not know exactly where the exception has been thrown. For example, it could be during the creation of the SytemUnderTest.

We can use the Execute Around Method Pattern with Lambda Expression to address this limitation. This is by creating a helper method, which accepts an Action<T> parameter as delegate.

```
public static class ExceptionAssert
{
  public static T Throws<T>(Action
```

```
  action) where T : Exception
  {
    try
    {
      action();
    }
    catch (T ex)
    {
      return ex;
    }

    Assert.Fail("Expected exception of
    type {0}.", typeof(T));

    return null;
  }
}
```

Now the text method can be invoked:

```
[TestMethod]
public void
SomeTestMethodThrowsException()
{
  var sut = new SystemUnderTest("some
  param");

  ExceptionAssert.Throws<Exception>(()
  => sut.SomeMethod());
}
```

The above ExceptionAssert<T>.Throws() can be used to explicitly invoke the method that throws the exception.

A separate note...

We would not have this limitation in some of the other Unit Testing framework such as NUnit, xUnit. These frameworks already have built-in helper methods (implemented using this pattern) to target the exact operation that cause exception.

For example xUnit.NET has:

```
public static T Throws<T>(Assert.
ThrowsDelegate testCode) where T :
Exception
```

Summary

In this article we looked at various .NET Design Patterns. Design Patterns are good but they only become effective if we can use them correctly. We would like to think Design Patterns as a set of tools which allow us to make better decisions on the code base. We would also like to treat them as communication tools so we can improve the communication around the code base.

We have looked at the *Abstract Factory Pattern* and *Cascade Pattern*. We have also looked at applying a slightly different approach to the existing Design Patterns using lambda expressions. This includes the *Cascade-Lambda Pattern*, *Pluggable Pattern*, and finally the *Execute Around Pattern with Lambda Expressions*. Throughout this article we saw that Lambda Expressions are a great way to enhance the power of some of the well-known Design Patterns ■

◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇

About the Author

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

WHY WOULD I USE SHAREPOINT AS A CONTENT MANAGEMENT SYSTEM?

This article provides an opinion from the point of view of a 30 year IT veteran about Content Management and SharePoint. The opinion is based on experience with SharePoint 2010 and takes SharePoint 2013 and 2015 into consideration. Too often we get caught up in numbers and versions. Numbers are all fine and good, but an opinion from practical experience is also important.

As with all individual statements, there are varying degrees of agreement and dis-agreement over the use of SharePoint as a Content Management System.

Most of these agreements and dis-agreements are based on varying opinion of SharePoint meeting a particular need in content management, and in a particular environment. This enforces the reality that for some, SharePoint as a content management system is absolutely one of the best products on the market; and for others it is total junk. Here in my opinion, I include the role a company's management plays in the success or failure of SharePoint as a content management system; along with issues and solutions Microsoft and 3rd parties provide in SharePoint for content management.

Basic Content Management System Defined

Before delving into SharePoint's capabilities (or the lack of) for content management, we need to define what exactly content management is. Breaking down the three words of the subject, we find a broad meaning.

- Content equates to any item which is held and managed by another item. A box manages its content by helping to prevent damage, helping to maintain organization, and in some cases even prevent unauthorized access.

- Management – that same box provides the ability to stack items in a location where the content is organized and allocated space is used in an optimum way.

- System – any methodology that aids in management of the content in the box, by use of the box in a consistent manner.

With this simple definition, it would appear that SharePoint is a content management system. However a typical content management system in modern day terms, goes far beyond this simple definition.

Modern Content Management Systems

Today a content management system must do more than just hold and provide simple organization of content. If that weren't true, then a simple network share would suffice as a content management system. The problem is the mess that develops in network shares, is typically what starts the call for a content management system. Even in a small business environment, a network share can quickly become very un-organized where multiple duplicate items exist and typically in multiple locations. A system is needed where items could be electronically checked out and secured from change except for the checked out version; and then checked in for others to use and make changes if needed. Today's content management systems provide:

- Organizing - content into storage folders or locations.

- Security controls to prevent un-authorized access, changes, and deletion.

- Publishing capabilities - holding drafts until the author or those authorized approve changes for those with read or greater access to view.

- Workflow - for automated routing, review and publishing.

These services are generally provided through a single interface such as a web page. This is starting to sound a little like collaboration using a centralized access point – sort of like SharePoint?

Stretching the boundaries of content management today include many types of websites that we are all familiar with. Facebook®, Twitter®, and many other social websites are all to some degree, content management systems. *Content management is the management of any type of content, the content management system was designed to handle. In my initial definition, I used a box. In modern terms, that box is managed by a computer application, a content management system full of tracking numbers, quantities, and so on.* In the broadest terms, everything we do from backups, to warehouse management, to document or data management is content management. That makes SharePoint a content management system by its design.

Getting Confused By Terms and Definitions

It's very easy to look at a document management system, asset management system, imaging management system, records management system, and many other "fill in the blank" management systems and say they are all content management systems. They are, but to varying degrees. These applications (combined) that manage content are referred to as *enterprise content management*. All of these systems are similar to a content management system, with the exception that they are specialized to manage a particular type of content. A document management system in most situations wouldn't work well to store project code in. The same works in reverse. Code management systems would not work well for a document management system.

Where does a content management system fit then? *I believe a content management system is a generic type of management system that can manage many different types of content.* A content management system can be made to do the work of a document management system, but is it as good as a dedicated document management system? For instance with SharePoint, is it a great document management system? In my opinion SharePoint is *not a great* document management system, at least not without moderate to heavy modification, but it is a *good* one. We're back to the original question, why use SharePoint for a content management system?

Content Management System - Breaking It Down

Let's break down what we expect out of a content management system and how SharePoint can or cannot fulfil those requirements. Remember, this is all based off of SharePoint 2010 (Enterprise) and is just my opinion, others may vary. I will mention where use of 3rd party tools allowed me to go outside of what out-of-the-box SharePoint was capable of, or the 3rd party tools simply did the job better than out-of-the-box SharePoint.

Data Storage – *Initial storage and the capability to expand storage capacity with any content management system is very important.* In most content management applications, storage isn't an issue. Content management systems have long been designed to have very large storage and expansion capability. In most cases it can be said that SharePoint has adequate capacity for content management because it has very good expansion capability. SharePoint does have hard numbers for sites, storage capacity of data in lists and document libraries, document size and so on. There are three real issues that I've seen come up with SharePoint as a content management system concerning data storage.

- The first is in document size. SharePoint under most circumstances is limited to 2GB file size or smaller. This isn't completely SharePoint's fault. A 32 bit pointer in SQL has been an issue for storage of large files in the content database for not just SharePoint, but for other systems using SQL for data storage.

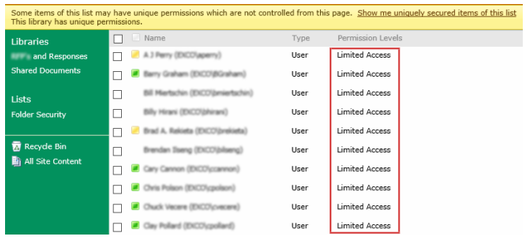
- There also is the maximum upload size in SharePoint. Setting the upload file size in SharePoint to its maximum can help but there's still a hard limit.

- The third problem is with speed, or the lack of it. SQL Server is a great database system. It responds quickly and has a great deal of storage capability. However, transferring data from SQL through SharePoint isn't what I believe anyone would call fast. One department I worked with uses massive datasets in the order of 200, 400, or even 500 gigabyte of data. The department manager was not pleased when I told her that yes SharePoint can store large amounts of data, but there are limitations and that it'll take many hours (like start the transfer before you go home and hopefully it'll be done by morning) to complete. SharePoint does not shine when trying to move large amounts of data. For a company that has a ton of data, moving it into SharePoint is very time consuming and difficult. Then getting that data back out brings up the complaint of "it's so slow". Add to that the SQL manager yelling about how the SQL log and audit file system filled up and is causing all kinds of havoc.

The solution we found that worked best is actually part of SharePoint – BLOB storage (Binary Large Object). We used Metalogix International's *StoragePoint*, a BLOB component for SharePoint. You can also use Microsoft BLOB storage. StoragePoint BLOB storage met our needs and has a few extra features we liked. *With BLOB, all data content is stored in an external container instead of the SharePoint content database.* In StoragePoint, pointers are placed in the SharePoint content database that point to the external content. Our StoragePoint BLOB is on our NetApp. Large data transfers are completed using the Explorer window in SharePoint and between our network shares, where the data is temporarily held. Transfers are much faster both to and from SharePoint, and limitations are reduced.

Security – *In some situations the security of data is critical. Who can read, write, and delete is a primary reason to have a content management system. The security in SharePoint is spectacular as far as granularity is concerned.* A site all the way down to a specific document or list item can have its own specific security setting. The issue is permissions are controlled manually and that becomes a very big headache. An example is that I have a document library in a site. In this document library I have 5 documents. Three of the documents can be read by anyone in the company. For the other 2 documents, I need to provide exclusive read capability to only a specific set of people. First I have to grant all the ability to see and open the site using read access. The same goes for access into the library. Under most situations, a document library will inherit the

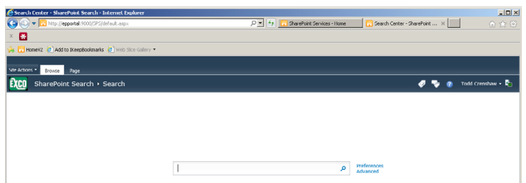
site security. Now for the 3 documents everyone can read, I can leave those alone allowing them to inherit the read only security from the library which was inherited from the site. For the 2 documents that I want limited read access to, I have a choice. I can create a folder and place those documents into that folder. I then break the permissions on that folder and apply new permissions on the folder. Or I can do the same for each document. You can see that if I only need to do this for a few documents, either method will work and isn't really a bother to accomplish. Now imagine the need to do this for 100 documents or 1000 documents. There's also another issue here – finding out who has permission to do what to each document, is crazy in SharePoint. At different security levels, you'll run into SharePoint telling you at a site level that John Smith has "Limited Access" to the sites content. What John Smith can access is much more difficult to determine. Limited Access means John Smith has access to something in the site – what he has access to is really fun to figure out (sarcasm).



Permissions in SharePoint as a content management system, especially dealing at a granular level of specific documents or list items, can be rather difficult and irritating. There are solutions though – several 3rd party applications will assist with security at granular levels. In my situation I found a 3rd party workflow add-on from *HarePoint* named Workflow Extensions. This add-on provides over 200 additional actions to the 30 out-of-the-box actions provided in SharePoint. Some of the actions available allow my workflow to change permissions at site, library, and individual list or even document levels. I simply write a workflow to allow site administrators to decide what level of security needs to be placed on a document or list item. They make a change to the item, the workflow runs, and the appropriate permissions are set.

Content Types – Capturing metadata and other information can be done fairly easily in SharePoint. *Content types have been around for some time in SharePoint and enforce continuity and consistency in content entry into a list or library. The drawback is that just about everyone hates content types.* The sad part is that it is an excellent way to capture data allowing a content management system to work well. What I've found is that when I enforce a content type, I usually get little use out of the library. I get no content because no one wants to fill out the required information. Because of this, I'm not a fan of using content types and enforcement of metadata in SharePoint. Getting your users to use it is very difficult and there are other ways to gather metadata and content information.

Search – I hear a lot of complaints in the SharePoint community about the out-of-the-box search in SharePoint. It's so generic and bland looking. Its advanced search is a joke, and other comments have been made. I'll admit that it's not the greatest search tool in the world, ok not even close, but it does work. I've not had many instances where I've gone looking for a document and not found it. *More important is that Microsoft provides SharePoint with one of the better search capabilities in Microsoft's FAST search server. With FAST it's virtually impossible to not be able to find what you're looking for in SharePoint. FAST resolves the "I can't find it in SharePoint" comments from end users.*



Versioning – *The versioning capability in SharePoint is fairly good, but I wouldn't call it excellent.* One thing versioning does do that I like, is display the list item or document in its true format before changes were made. This means I can open a prior version, open an existing version and display them side by side to see the changes. The bad news is that versioning doesn't highlight or in any way indicate, what was changed. You have to eyeball it. The versioning also can't tell the difference between a legitimate change and from a field that was changed, and then changed back to what was originally in that field. To SharePoint versioning, that was a change even though the list item or document is identical to the prior version.

Publishing – *SharePoint has the capability to hold a document or list item in a hidden or unpublished state, only displaying an approved and published version of that document.* This provides a review process prior to providing employees with information that they may not be authorized to view. By default, SharePoint publishing is turned off. Once turned on, an approval process is setup and followed for each item in a list or document in a library.

Workflows – Content Management Systems almost always have some sort of workflow engine. SharePoint has a workflow approval process built in. It is limited though. This is an area where some content management systems fail, including SharePoint. *The OTB workflows in SharePoint provide some good workflow actions but they are still limited.* Add-in 3rd party workflow extensions and much more can be accomplished. For instance I mentioned above about the ability of the workflow extension to address permissions on list items and documents. Additional capabilities include adding and removing users from appropriate AD groups for access; converting documents from one format to another, copying or moving a document to an appropriate document library, checking in a stale checked out document, sending the document to an FTA server or website, or even modifying document content based on rules in the workflow. Workflows are the power behind a good content management system, along with someone capable of working complex workflows. *To be honest, without the HarePoint Workflow Extensions I use, I would not be able to provide the routing, approval, and security requirements needed to meet the requirements of my customers for a content management system.*

Upgrade – A good content management system should be upgradable. Take into consideration a system that contains most company documents. Management of those documents has been through the use of workflows like what I've mentioned in the prior section. Just mentioning upgrading SharePoint (let's say from SP 2010 to SP 2013) where workflows are extensively used to a SharePoint admin will turn them white. *Upgrading a content management system should not break the processes put into place to manage that content. Unfortunately that's typically what happens with a SharePoint upgrade.* This area is of great concern if you're looking at a content management system and SharePoint is in the running. I've already mentioned that moving large amounts of data in SharePoint isn't always easy or fast. Since workflows will most likely break when converted to the new version of SharePoint, the alternative is to setup and build an entire new fresh SP 2013 (or beyond) system. Once done, move the data over to the new system. This is time consuming, tedious, and what do you do for the next upgrade?

Management Support – This is key for any content management system, not just SharePoint. My experience with SharePoint in an attempt to move data from an Engineering departments network share into SharePoint, bombed terribly. There are two reasons for this. *The first is that management hired a content management expert who had no idea how SharePoint handles content.* This person knew one way and that was through the use of network shares. Shares and files were setup to use a cryptic naming convention that is not compatible with SharePoint. Many attempts were made to bring this person into the SharePoint environment and method of content management with no luck. The second issue you may run into had to do with management. *The management that wanted to move the network share data into SharePoint was not committed to moving the data.* Instead the management decided that the content management person was right and there was no possible way to move the data into SharePoint. The project to move the data into SharePoint went on for almost two years with almost no progress, until it was finally dropped. *Without management pushing for real content management, no matter what system is purchased, SharePoint or any other, success will not be met.*

Summary – The big picture questions:

- Do you need a content management system, a collaboration area, a dedicated content management system like a document management system, or a combination of the above?

- What are your storage requirements now and in the future? This should include both capacities as well as file sizes.

- What are your security requirements going to be now and in the future? Is this system going to need to be SOX compliant?

- What will the requirement be for metadata, indexing, grouping of similar documents and is that going to be by document type or content?

- What is the importance of versioning for your organization? Will original versions of documents stored 7 or more years ago be required?

- How well does the search capability of the chosen content management system work?

- What types of publishing controls are needed and of great importance, who will control and approve published documents?

- Will automated routing be needed or other controls that workflows can provide to enhance and ease management of content, and who will write and maintain those workflows?

- How often will the chosen system need to be upgraded and will upgrading cause any issues?

- **Does your management support the use of a content management system?**

Out of all of the above, the most important is the last bullet item. Without support of management; and this is more than just management agreeing to fund the project, a content management system is hard bound not to fail. Management must insist upon the use of a content management system and remove alternatives.

The good news is that for a content management system, SharePoint can meet most of the above requirements fairly easily. Yes add-on features will most likely be needed, and there are drawbacks such as difficulty in upgrading. However for an overall system to store and manage multiple different types of data, SharePoint can provide adequate services for most small to mid-size companies.

Conclusion – For a small business just getting into content management, SharePoint will work well, even the standard version without 3rd party add on's. For a mid to large business, moving to SharePoint for content management, first and most importantly must have management backing and support. Management must also be willing to provide the tools to enhance SharePoint such as BLOB storage, 3rd party workflow extensions, FAST search, and education of employees on the reasons behind such things as content types and their importance. Isn't management support a key factor in almost all business choices? SharePoint as a content management system is no different ■

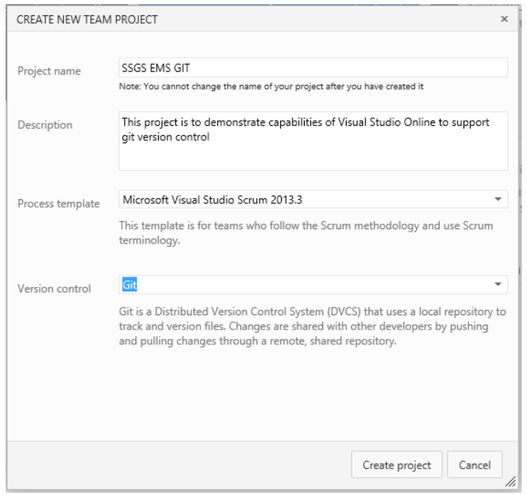
About the Author

Todd Crenshaw is a Systems and SharePoint Administrator for an energy company located in Dallas, Texas. Todd has worked in IT for 30+ years and has experience covering all Windows platforms, several flavors of UNIX, and IBM's AS/400 series. On the application side, Todd's managed SharePoint 2007/10, Exchange 6.x, several different flavors of enterprise backup software. Over the past 6 years Todd has been involved with SharePoint in the administration and Power User roll where he has pushed the limits of SharePoint workflows and InfoPath. Todd can be reached through LinkedIn and maintains his own SharePoint blog at here

Git is a Distributed Version Control System that was developed initially as part of the Linux development initiative, and later on as an independent open source project. Microsoft Team Foundation Server embraced Git from its version TFS 2012.2. While creating a team project either on 'on-premise' TFS or Visual Studio Online Now, Git is one of the options available along with traditional Team Foundation Version Control (TFVC).

VISUAL STUDIO ONLINE WITH GIT

Image courtesy: visualstudio.com

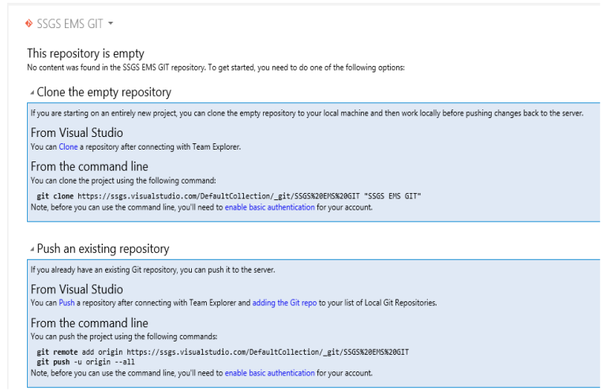


Team Foundation Version Control (TFVC) is a centralized version control system. It stores all the versions of software at a central repository; the database of TFS. Whenever a request comes in, either the latest version or a specific version of files is given to the user by transferring over the network from TFS. All the other versions remain on the central repository only.

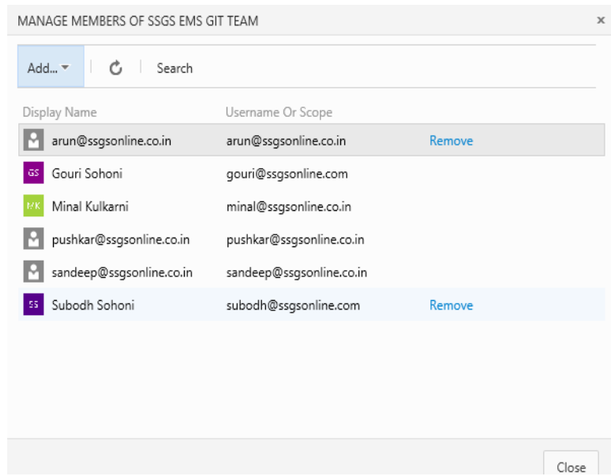
Behavior of Git is a little different from that of TFVC. Whenever the user wants to work on a software and has permission to do so, Git forces the user to clone the server repository on the user's machine. In this way, the user gets all the versions of all the files of that software in a local repository.

The User now can work on all the versions of that software locally, which of course is much faster than working with versions that are located on a central server somewhere in the network or the internet. So far, this model is excellent as far as an individual developer is concerned, but modern nontrivial software development is a team work. How will the developer share his / her work with team members in this model?

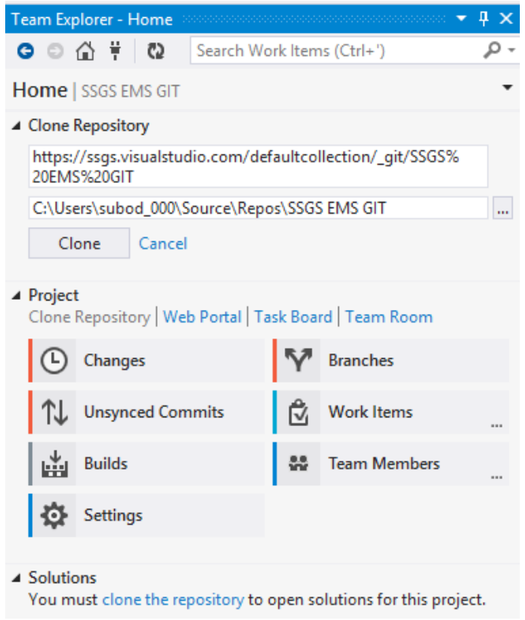
This is where TFS or Visual Studio Online (VSO) comes in. When a team project is created with the option of Git selected, a server side empty Git repository is created. Developers are suggested to clone that empty repository using Visual Studio or a command line tool (Third party tools are available for free download). The image shows the page that is shown to the user when he / she clicks the CODE menu on the TFS Web Interface.



Whenever the developer needs to share work with others, he / she pushes that part of the software (files) on the central Git repository that is kept on the TFS or VSO. For this to work, all those team members have to be member of the *contributors* group of that Team Project.



If a new user joins the team, he / she will start by cloning the remote repository that exists on VSO for that project. Files shared by all earlier developers with all the versions of those files will be then available to that developer in the local repository.



Before we dive deep into the concepts of Git as implemented on TFS / VSO, a few words about VSO itself. VSO is an implementation of TFS on the cloud maintained by Microsoft. The benefits of VSO that I can think comparing it to on premise TFS, are numerous:

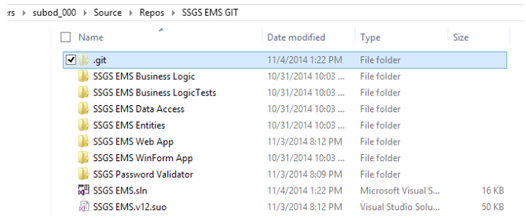
1. You need not implement and maintain TFS itself. It is maintained by Microsoft. This reduces your infrastructure cost as well as efforts to back-up database for failovers.

2. Latest features of TFS are always available on VSO much earlier than same features being made available on on-premise TFS.

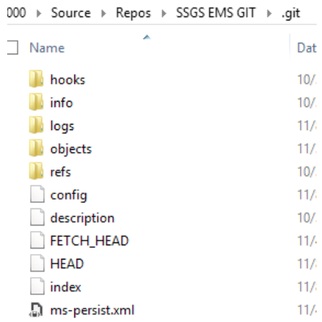
3. VSO is location transparent. Since VSO is in the cloud, it does not matter from where the team members are accessing the services of VSO.

For a free trial of VSO, Microsoft offers a free account that can have a team of upto 5 users who do not have MSDN subscription. Any team members having MSDN Subscription are considered in addition to these 5 accounts.

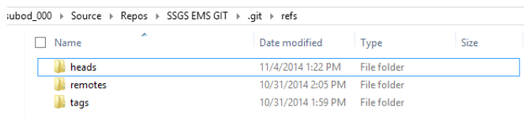
When a repository is cloned it creates a folder on the user's machine that contains the files and a hidden folder named .git.



This .git folder contains all the settings, logs, hooks and the source objects on which you are locally working.

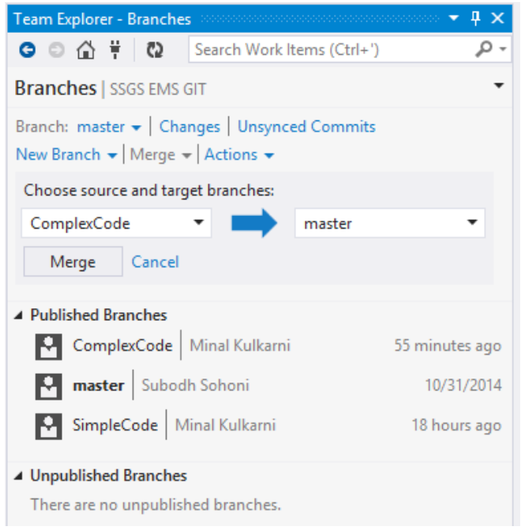


.git has a subfolder named "refs" that contains the named references to the branches in a subfolder "heads" and references to the remote branches that you have "fetched" in the subfolder "remotes".



We came across an operation of git in the earlier statement named "fetch". "fetch" is an operation that brings the latest updates from remote branches that you are tracking. These remote branches are shared by other developers on VSO. Fetching a branch from remote repository does not mean you can still work on it and edit code in it. If you too want to participate in contributing in the same branch, then you need to create a local branch that clones the remote branch. Once that cloned local branch is created, you can work on that branch, create new code in it or edit existing code. After doing these edits, you can "commit" them. "Commit" operation is very similar to Check-in of TFVC but committed code does not go to the server as the checked in code does in TFVC. It remains in the local repository only, i.e. in the branch that you are working right now. That branch's last commit is called "HEAD", note the difference in the case. "head" is a named reference of any branch whereas "HEAD" is the variable that stores the latest commit of the branch that is active.

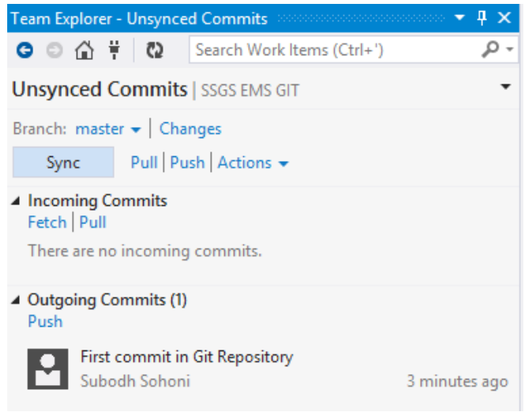
Before you commit the edited code, as a best practice you should fetch and merge the remote commits in your code. Merge is an operation that merges the code from one branch into the other.



VSO provides another option that combines the "fetch" and "merge" operations and that is called "pull". Pull operation does an automatic merge immediately after fetch operation on a remote branch. Many experts still prefer the two separate operations to be done manually since it provides a degree of control on what is merged and how the conflicts are resolved. It is the same as TFVC which can do automated conflict resolution but does also provide a manual option to do so.

I do also recommend doing manual merge unless you are in too much hurry and there are too many changes to merge (and you have a nature that accepts inherent risks of automated conflict resolution!).

You may commit code many number of times. Each time it will move the tip of the branch to the latest commit so that it will become the HEAD for the time being. After you have committed the code that you are editing and are sure that you may now share the code with others, you can "push" the code to the VSO server repository that is shared by the entire team.

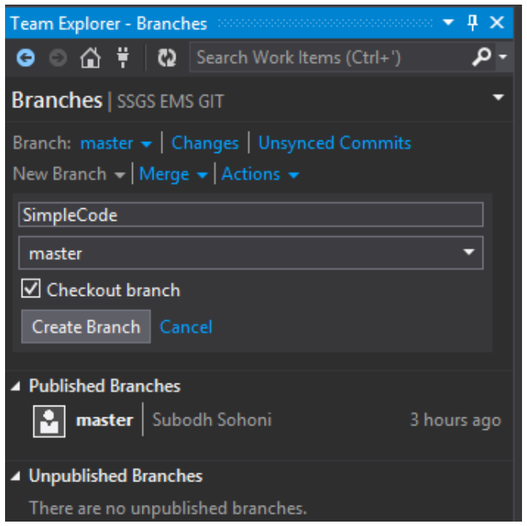


The Sync button that you see in the image is an operation specific to VSO (& TFS) that is a combination of "pull" which of course includes "fetch" with "merge" and "push" operations. It synchronizes the local branch with the corresponding remote branch.

After "push" operation, VSO will reflect that on the CODE option of the team project and will also show the details of that operation - like when was the last change (commit) made, what was the comment given by developer while committing that and the name of the developer who committed.

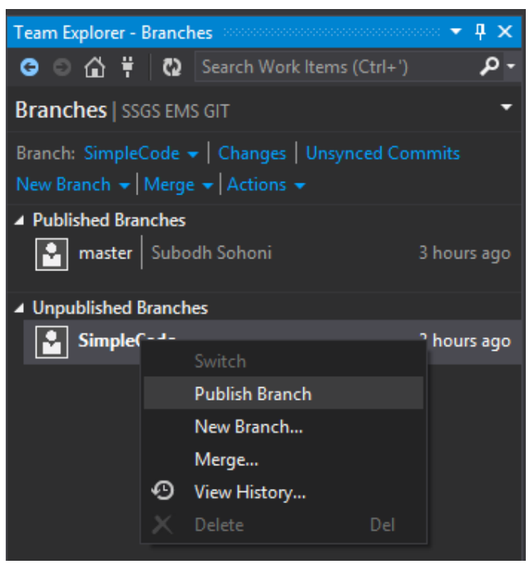
When the Git repository is created by VSO for a team project and the team starts working on it, there is only one branch that is created by VSO. By default it is named as *Master*. Whenever you want to work on some code that needs special attention, at that time you can create another branch from Master.

If and when other branches exist, you may also branch out from them. For example, a team member wants to try out some code before it can become part of the release. She creates a branch called SimpleCode for this purpose.

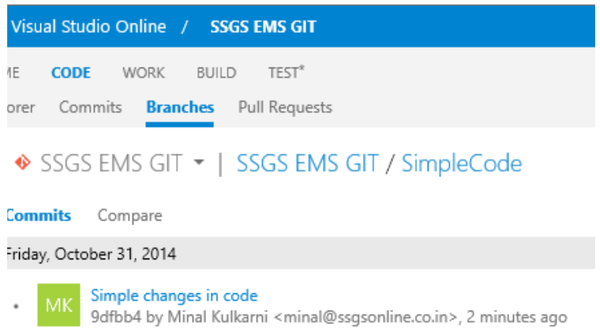


The check-box for "Checkout branch" that you see in the image is not the Checkout operation that is commonly known to the developers who are using TFVC. In git "Checkout" operation is to make that branch as the active branch, so that HEAD becomes the tip of that branch.

After the branch is created, VSO keeps it as a sort of private branch, unpublished to the other team members. To share that branch with the team members, you need to publish that branch which you can do by using the context menu (right click) of the unpublished branch.



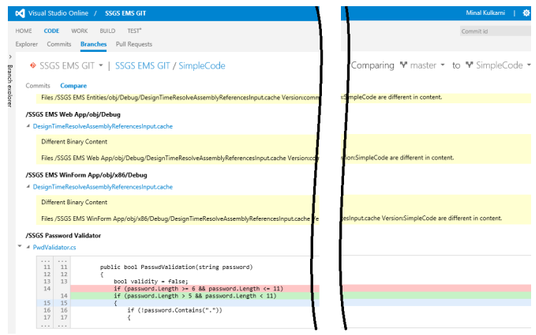
Now that branch is shown on VSO under the Branches item under CODE section of the team project. It also shows the commits done in that branch.



Over the time many branches will get created in the repository. VSO does not yet have a hierarchical visualizer for git branches that we are accustomed to in TFVC. It makes up for that by providing the relative status of branches in comparison to a specific branch. For example in the image shown here, branch ComplexCode is selected by a user. Commits that are done in other branches as well as the ones that those do not have, are shown.

The branch SimpleCode is "Behind" ComplexCode branch by 2 commits which means there are 2 commits in ComplexCode that are not merged in SimpleCode and at the same time it is "Ahead" of ComplexCode branch by 1 commit. It has one commit that is not merged with ComplexCode branch. Status of each branch is shown in this way. If you change the selected branch, then the status also changes to show relative commits.

You may also compare two branches to check the differences in those branches. It not only shows which files are different, but VSO also shows the code that is different in two branches.



These features that are under the Branches tab of CODE section of VSO are not available when you use TFVC. I found these features to be excellent and worth making available in TFVC also.

TFS allows some settings to be done on Git by individual developers. They can provide their name, email and default location of repository.

Ignore File (.gitignore) is a generic Git setting that allows the specified file types to be ignored in commits by Git. When this file is created using Visual Studio, it adds by default the extensions for Solution Files, Built Assemblies, Test Results etc. Similarly another file (.gitattributes) can be added that stores the attributes required by Git commands.

There are certain unique features in TFS Version Control (with TFVC) that are found to be missing in Git with VSO / TFS. First and foremost is a glaring absence of Check-in policies. Although hooks can be developed for client side checks, it does not replace the server side hook that can be a shared rule to be observed when developers push their code. I did not find any documentation to develop these hooks. Another noticeable absence is that of Gated Check-in. Such unique features provide better value to TFS / VSO in comparison to other tools and I hope that Microsoft provides these as features in the near future.

Summary: For a developer who has always used centralized version control system, the concepts of distributed version control system like Git are very different. Git is becoming popular due to the speed that it provides to developers to do version control operations. TFS and VSO have embraced Git as one of the default options for version control while creating a new team project. In this article, I have tried to put the concepts of Git as implemented by TFS / VSO in perspective for a developer who is accustomed to TFVC ■

About the Author

Subodh Sohoni, Team System MVP, is an MCTS – Microsoft Team Foundation Server – Configuration and Development and also is a Microsoft Certified Trainer(MCT) since 2004. Subodh has his own company and conducts a lot of corporate trainings. He is an M.Tech. in Aircraft Production from IIT Madras. He has over 20 years of experience working in sectors like Production, Marketing, Software development and now Software Training. Follow him on twitter @subodhsohoni and check out his articles on TFS and VS ALM at http://bit.ly/Ns9TNU

ServiceStack is an independent, self-sufficient, light-weight framework built on top of ASP.NET for building robust end-to-end web applications.

Exploring ServiceStack

Security, Bundling, Markdown views and Self-Hosting

In the previous edition of the DNC .NET Magazine (Issue 15th Nov-Dec 2014), we introduced Service Stack and saw how easily one can build web apps and web APIs using this great framework. The same article can also be accessed online over here.

If you haven't read the previous article yet, I would encourage you to read it and check the sample code before continuing further. Code sample of the previous article contains a Student Reports application that performs basic operations on a database and shows data on Razor views. In this article, we will continue adding the following features to the same sample:

- Authentication and authorization
- Bundling and minification
- Markdown views

..and we will briefly discuss how to create a self-hosted ServiceStack application.

Authentication and Authorization

Security is an integral part of any enterprise application. Every framework that allows us to create great apps should also provide ways to secure the app for legitimate users. ServiceStack comes with an easy and effective solution for applying security on its components.
In the period of last 2 years or so, OAuth has gotten a lot of attention. We see a number of popular public sites allowing users to authenticate using their existing accounts on Google, Facebook, Twitter or a similar site. ServiceStack offers this to us for free of cost; with minimum setup required to get it up and running.

Authentication Providers and Repositories

To enable authentication and authorization in a ServiceStack app, we don't need to install a new NuGet package to the app. The feature is included in core package of the framework. The framework includes a number of auth providers; following are most commonly used ones:

- CredentialsAuthProvider: Authenticates based on Username and Password
- AspNetWindowsAuthProvider: For windows authentication
- BasicAuthProvider: Provider for Basic authentication
- DigestAuthProvider: Provides Http digest authentication
- TwitterAuthProvider: OAuth provider to authenticate a user with Twitter account
- FacebookAuthProvider: OAuth provider to authenticate a user with Facebook account
- GoogleAuthProvider: OAuth provider to authenticate a user with Google account

We can write our own custom authentication provider by either implementing *ServiceStack.Auth.IAuthProvider* interface or by extending *ServiceStack.Auth.AuthProvider* class. It is also possible to write our own OAuthProvider by extending the class *ServiceStack.Auth.OAuthProvider.*

In the sample application, we will add credentials based authentication and Twitter authentication. Let's start with credentials authentication.

Credentials Authentication

To enable credentials authentication and to provide users to register, we need to add the following statements to Configure method in AppHost.cs:

```
Plugins.Add(new AuthFeature(() => new
AuthUserSession(), new IAuthProvider[]{
  new CredentialsAuthProvider()
  }));
Plugins.Add(new RegistrationFeature());
```

The *CredentialsAuthProvider* needs a repository to store/retrieve user information. The framework has

a set of built-in repositories that work with some of the most widely used data storage mechanisms. Following are the repositories available:

- *OrmLiteAuthRepository:* To work with an RDBMS to manage user data
- *RedisAuthRepository:* To use Redis as data source containing user information
- *InMemoryAuthRepository:* Stores user information in main memory
- *MongoDBAuthRepository:* To use MongoDB NoSQL DB as data source containing user information
- *RavenUserAuthRepository:* To use RavenDB NoSQL DB as data source containing user information

These repositories serve most of the common cases, and you can write your own repository by implementing *ServiceStack.Auth.IUserAuthRepository*. In the sample application, since we are already using SQL Server for application data; let us use the same database for storing user data. So *OrmLiteAuthRepository* is our obvious choice.

The *OrmLiteAuthRepository* class is defined in *ServiceStack.Server* assembly. Let's add it using NuGet.

```
Install-package ServiceStack.Server
```

The *OrmLiteAuthRepository* needs an *IDbConnectionFactory* object to perform its action. All we need to do to use *OrmLiteAuthRepository* is create an object with an *IDbConnectionFactory* and register it with the IoC container of ServiceStack, *Funq*. Following are the statements to be added to *Configure* method to perform this task:

```
var repository = new
OrmLiteAuthRepository
(ormLiteConnectionFactory);
container.
Register<IAuthRepository>(repository);
```

The *OrmLiteAuthRepository* instance needs some of tables to work with. As we don't have the tables already created in our DB, we can ask the repository to create them for us. For this, the repository has a method called *DropAndReCreateTables*. As the name suggests, it would drop and re-create all the tables. So we need to add a condition to check if one of the tables is already created. This might not be the best possible check, but works for the demo.

```
using (var db =
ormLiteConnectionFactory.Open())
{
  if(!db.TableExists("UserAuth")){
    repository.DropAndReCreateTables();
  }
}
```

Exploring Authentication Endpoints

Now if you run the application and see metadata page of ServiceStack, you should be able to see some new endpoints automatically added for us.

Figure 1: New Endpoints

Let's register a user using the endpoint. Open Fiddler or, Chrome's REST Client plugin or any HTTP debugger of your choice. I am use Fiddler to play with the APIs I build. We will send a POST request to the Register endpoint listed in Figure-2.
To know how to use this endpoint, click the JSON link (You can click any of the corresponding links; we will use the JSON endpoint in our app). It shows the options and format of the data to be posted to the endpoint.

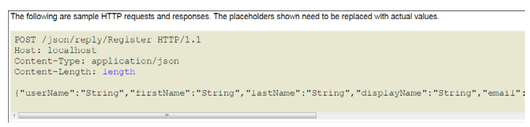


Figure 2: Data format and options

In your HTTP debugging tool, compose a POST request following the structure in Figure 3. Figure 4 shows how to do it in Fiddler.

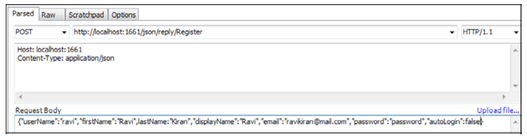


Figure 3: POST Request

After executing this request, you would get the following response indicating successful registration from the server:

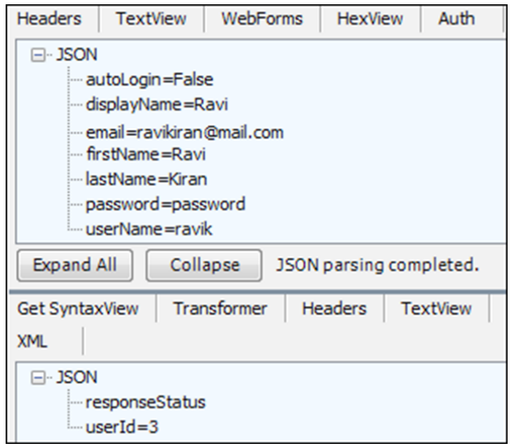


Figure 4: Response in Fiddler

If you check the UserAuth table now, it has a row with the data you just entered:

Figure 5: Newly added user in UserAuth table

If you set *autoLogin* to true in the JSON data sent with the request, you would also receive a session ID along with the request. I intentionally didn't do it so that I can show how to login using the auth API.
To login using the API, we need to send a POST request to /auth/credentials endpoint (as we are using credentials provider). You can check the request specification in the documentation. Following is the request sent to login API from Fiddler:

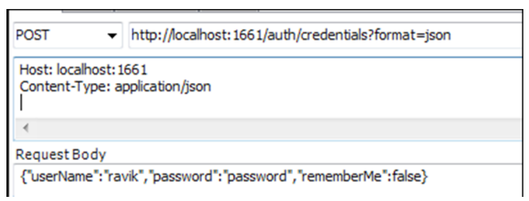


Figure 6: POST Request for login

If your credentials are correct, you will get a success response from server containing session ID.

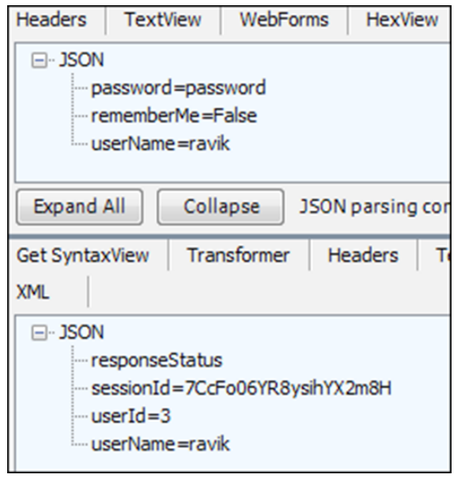


Figure 7: Successful response with SessionID

Similarly for logout, we need to send a request to the following URL to get the user logged out of the app: http://localhost:<port-no>/auth/logout?sessionId=<sessionId>

Applying Authentication on a page and Creating Login Form

Let's impose authentication rules on the Marks API. It can be done by simply adding Authenticate attribute to the *MarksRequestDto* class.

```
[Authenticate]
public class MarksRequestDto
{
    //Properties of the class
}
```

Now if you try to load the marks page of any student, you would be redirected to the login page. As we don't have a view for login yet, the browser will display an error. Let's add a simple login page to the application. We need a DTO class and a service to respond to the request.

```
public class LoginService : Service
{
  public object Get(LoginDto request)
  {
```

```
    var session = this.GetSession();
    if (session.IsAuthenticated)
    {
      var redirectionUrl = (request.
      Redirect == string.Empty ||
      request.Redirect == null) ?
      "students" : request.Redirect;
      return this.
      Redirect(redirectionUrl);
    }

    return request;

  }
}

[Route("/login", Verbs="GET")]
public class LoginDto
{
  public string Redirect { get; set; }
}
```

Add a view and name it LoginDto.cshtml. Add the following mark-up to the page:

```
@using StudentReports.DTOs
@inherits ViewPage<LoginDto>

@{
  ViewBag.Title = "Login";
}

<div class="row">
 <div class="panel panel-default">
  <form id="loginForm" action="/auth/
  credentials?format=json"
  method="post">
   <table class="table">
    <tbody>
     <tr>
      <td>Username: </td>
      <td> <input type="text"
      name="userName" /> </td>
     </tr>
     <tr>
      <td>Password: </td>
      <td> <input type="password"
      name="password" /> </td>
     </tr>
     <tr>
      <td>
       <input type="submit"
       value="Login" />
      </td>
      <td>
       <input type="reset" value="Reset"
/>
      </td>
     </tr>
    </tbody>
   </table>
  </form>
 </div>
</div>

@section Scripts{
  <script src="~/App/loginPage.js">
  </script>
  <script>
    $(activateLoginPage);
  </script>
}
```

This page needs some JavaScript to handle login. Add a JavaScript file to the application inside a new folder App and name it loginPage.js. Add the following script to this file:

```
(function (window) {
 function activateLoginPage() {
  $("#loginForm").submit(function (e)
  {
   var formData = $(this).serialize();
   var loginUrl = $(this).
   attr("action");
   $.ajax({
    url: loginUrl,
    type: "POST",
    data: formData
   }).then(function (data) {
    location.replace(decodeURIComponent
    (location.href.substr(location.href.
    indexOf("=") + 1)));
   }, function (e) {
    console.log(e);
    alert(e.statusText);
   });

   e.preventDefault();
  });
 }

 window.activateLoginPage =
 activateLoginPage;
}(window));
```

Build and run your application after saving these changes. Change the URL to http://localhost:<port-no>/login. You will see the login page appearing on the screen. Login using the credentials you used to register; it should take you to home page after logging in. And now you should be able to view marks of students as well.

Note: *I am not creating the Register form as part of this article. Creating a register form is similar to the Login form except the fields and API to be used. I am leaving this task as an assignment to the reader.*

Adding Twitter Login

Let's add an option for the users to login using their twitter accounts. For this, you need to register your app on twitter app portal and get your consumer key and consumer secret. Once you get the keys from twitter, add following entries to your app settings section:

```
<add key="oauth.twitter.RedirectUrl"
value="http://localhost:<portno>/
students"/>
<add key="oauth.twitter.CallbackUrl"
value="http://localhost:<portno>/auth/
twitter"/>
<add key="oauth.twitter.ConsumerKey"
value="<your-twitter-consumer-key>"/>
<add key="oauth.twitter.ConsumerSecret"
value="<your-twitter-consumer-secret>"/>
```

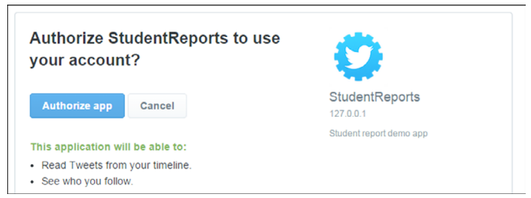
Names of keys shouldn't be modified. Finally, you need to modify the Authentication feature section in AppHost.cs as:

```
Plugins.Add(new AuthFeature(() => new
AuthUserSession(), new IAuthProvider[]{
      new CredentialsAuthProvider(),
      new TwitterAuthProvider(new
AppSettings())
}));
```

Now, we are all set to use twitter authentication. Save all files, build and run the application. Open a browser and type the following URL:

http://localhost:<port-no>/auth/twitter

This will take you to twitter's login page and will ask you to login. Once you are logged in, it will take you to the following page.



If you click authorize app in this page, it would take you to the home page and you will be able to access the pages that require authentication.

Similarly, you can add any other OAuth provider to authenticate a user. You can learn more about OAuth providers in ServiceStack on their wiki page.

Login/Logout link and Displaying Username

To logout, you need to send a request to the following path:
http://localhost:<port-no>/auth/logout API.

This API removes user details from the session. To make it easier, you can add a link on your page. Adding a login/logout link with name of the logged in user needs a bit of work. This is because we get user information from session in the service. We can read the user's name from session and pass it to view through response DTO. Following snippet shows how to read username and assign it to the response DTO:

```
public object Get(MarksRequestDto dto)
{
  var username = this.GetSession().
UserName == null    ? "" : this.
GetSession().UserName.ToTitleCase();


  //Logic inside the method

  return new MarksGetResponseDto()
  {
    Id = student.StudentId,
    Name = student.Name,
    Class = student.CurrentClass,
    Marks = new List<Marks>() { marks },
    Username = username
  };

}
```

In the Layout page, you can use this property to show username and login/logout link. Following is the snippet:

```
@if (Model.Username != “” && Model.
Username != null)
{
  <span>Hi @Model.Username!</span>
  <a href=”/auth/logout” style=”color:
  papayawhip;”>Logout</a>
}
else
{
  <a href=”/login” style=”color:
  papayawhip;”>Login</a>
}
```

If you login using local account it would display your local username, and if you use an OAuth provider, it would display username of your social account.

Authorization

Authentication makes sure that the user is known to the site and authorization checks if the user belongs to the right role to access a resource. To restrict users accessing a resource based on the roles, we need to apply the *RequiredRole* attribute on the request DTO class. Following snippet demonstrates this:

```
[Authenticate]
[RequiredRole(“Admin”)]
public class MarksRequestDto
{
  //properties in the DTO class
}
```

Apply this attribute to the *MarksRequestDto* class. Now if you try accessing the marks page for any student, the app won't allow you (and it redirects to home page because of the changes we did to login page). You can assign role to a user using */assignroles* API, but this API is restricted to users with Admin role. So let's create a new user with admin role when the application starts. Add the following snippet inside *AppHost.cs* after creating the tables:

```
if (repository.
GetUserAuthByUserName(“Admin”) == null)
{
  repository.CreateUserAuth(new
  UserAuth()
  {
    UserName = “Admin”,
    FirstName = “Admin”,
    LastName = “User”,
    DisplayName = “Admin”,
    Roles = new List<string>()
    {RoleNames.Admin }
  }, “adminpass”);
}
```

Login using these credentials and now you should be able to access the marks page.

A user can be assigned with a set of permissions too and the permissions can be checked before allowing the user to access an API. To set permissions to the admin in above snippet, assign a string list to the Permissions property:

```
repository.CreateUserAuth(new UserAuth()
{
  UserName = “Admin”,
  FirstName = “Admin”,
  LastName = “User”,
  DisplayName = “Admin”,
  Roles = new List<string>()
  {RoleNames.Admin },
  Permissions = new List<string>() {
  “AllEdit” }
}, “adminpass”);
```

Following is the attribute to be applied on the API to check for the permission:

```
[RequiredPermission(“AllEdit”)]
```

Now you can assign a role to an existing user. To do that, login using the admin credentials and invoke the /assignroles POST API with following data:

```
{“username”:”ravi”, “roles”:”Admin”}
```

After this step, you will be able to access the marks page when you login using ravi as username.

Bundling and Minification

In rich client based applications where you have a lot of client side script to be loaded during page load, loading time of the page gets affected because of number of round trips made to server to load all files. Using techniques like bundling and minification, these files can be combined and shortened to reduce the total size and number of downloads from server to make the page load faster.

ServiceStack supports bundling and minification. To enable bundling in the ServiceStack app, we need to install the *Bundler* Nuget package.

- Install-package Bundler

This package adds a new file Bundler.cs and a folder bundler to the project. If you expand bundler folder, it contains a node_modules folder and some executable cmd and exe files. This may pop a question in your mind that why would we need Node.js modules in a .NET application? The bundler.js file in this folder requires Node.js code to use these modules and perform the operations. This package is capable of other things including compiling LESS to CSS and CoffeeScript to JavaScript.

We have two files in App folder that we need to combine and minify. We need to create a .bundle file listing the files to be bundled. Add a new file to App folder and rename it to app.js.bundle. Add following content to this file (list of js filed to be bundled):

```
loginPage.js
marksRequestPage.js
```

By default, the package looks inside Content and Scripts folders for bundle files, but we can change it in the bundler.cmd file. Open the bundler.cmd file and modify the node command as follows:

```
f “%*” == “” (
    node bundler.js ../App
)
```

Now if you run the cmd file, you will get two files:

- app.js: Contains combined content of the files mentioned in the bundle file
- app.min.js: Contains combined and minified content of the files mentioned in the bundle file

From now on, whenever you make a change to one of these files, you need to run the bundle.cmd file. This step seems tiring; so let's automate it with the build process. Open properties of the project and specify following command under Build Events -> Post build event commandline:

```
“$(ProjectDir)bundler\bundler.cmd”
```

From now on, whenever you build the project, bundler.cmd would also run and it would produce two files in the App folder. One of these files would contain concatenated code from both files and the other would contain concatenated and minified code. To refer to the resultant file of the bundle, specify the following statement in Layout.cshtml after reference of jQuery:

```
@Html.RenderJsBundle(“~/App/app.
js.bundle”, ServiceStack.Html.
BundleOptions.MinifiedAndCombined)
```

The *RenderJsBundle* extension method is added in Bundler.cs file. You can play around with other options available in the enum *BundleOptions* by changing in the code and learn its behavior.
When you run the code after this change, you will see the above line gets replaced with a script tag similar to the following:

```
@Html.RenderJsBundle(“~/App/app.
js.bundle”, ServiceStack.Html.
BundleOptions.MinifiedAndCombined)
```

Markdown Razor View Engine

Markdown is tag less HTML. Meaning, you can compose HTML using plain text by following a set of conventions without writing even a single HTML tag. It is widely used these days to compose README files, post rich text on forums like Stack Overflow and even to write professional blog posts. If you have never tried markdown, check stackedit.io. The welcome page of the site contains a nice example of markdown covering most of the syntax. You can use HTML to apply any formatting that is not available in markdown.

ServiceStack's razor view engine supports Markdown views. You can use model binding syntax that you use in regular Razor pages and compose HTML with no tags. Add a new page to Views folder and name it AboutDto.md. Add the following code to it:

```
<link href="Content/bootstrap.min.css"
rel="stylesheet" />
<div class="navbar navbar-inverse
navbar-fixed-top" style="background-
color:darkslategray;">
  <div class="container">
    <h1 style="color:azure;">Student
    Marks Reports of XYZ School</h1>
  </div>
</div>
<section class="content container"
style="margin-top: 90px;">

## About Student Reports

This app is to view and manage reports
of students of XYZ School.

*Contact school management for any
questions on report of your kid.*

<br />
<br />
<br />

This page was last updated on @Model.
ModifiedDate by @Model.Name

</section>
```

Notice that we are using HTML just for applying bootstrap styles. Content of the page is free from tags. In the last statement, we are using a couple of properties from the model object. We can create a layout page for markdown and use it as base template for all markdown files. It is important to remember that Razor layout pages cannot be combined with markdown views and vice versa. We need DTOs and service for the view. Following snippet contains the code of these classes:

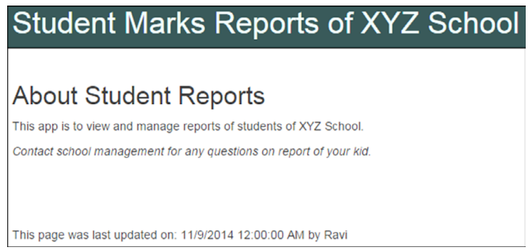
```
[Route("/about")]
public class AboutDto
{

}

public class AboutResponse
{
  public string Name { get; set; }
  public DateTime ModifiedDate { get;
  set; }
}

public class AboutService : Service
{
  public object Get(AboutDto dto)
  {
    return new AboutResponse() { Name =
    "Ravi", ModifiedDate = DateTime.Today
  };
 }
}
```

Build and run the application and change the URL to http://localhost:<port-no>/about. You will see the following screen in your browser:



Self-Hosting ServiceStack Apps

As stated in the first article, ServiceStack can be hosted on IIS, on a standalone application or even on mono. We don't need to make any significant change in the way we write code to build a self-hosted ServiceStack application. The difference is only in the way we define the AppHost class.

Create a new Console application and install the *ServiceStack* NuGet package to it. Add a new class and change the name to *EchoService*. Add the following code to this file:

```
public class EchoService: Service
{
  public object Get(EchoDto echoObj)
  {
    return "Hello, " + echoObj.Name;
  }
}

[Route("/echo/{Name}")]
public class EchoDto
{
  public string Name { get; set; }
}
```

It is a simple service that takes a name and echoes a hello message. To host this service, we need an *AppHost* class. Add a new class to the application and name it *AppHost*. Add the following code to it:

```
public class AppHost : AppSelfHostBase
{
  public AppHost():base("ServiceStack
  hosted on console app",
  typeof(EchoService).Assembly)
  {
  }

  public override void Configure(Funq.
Container container)
  {
  }
}
```

The difference between *AppHost* in web application and Self-hosted application is the base class of *AppHost* class. Here, we have *AppSelfHostBase* instead of *AppHostBase* as the base class. I will leave the Configure method empty here as we are not going to add a lot of logic to this application.
Now the only thing left to do is starting the server on a specified port. To do this, add the following code to the *Main* method in the *Program* class:

```
static void Main(string[] args)
{
    var portno = 8888;
    new AppHost().Init().Start(string.
Format("http://localhost:{0}/",portno));

    Console.WriteLine("ServiceStack
started on port no: {0}. Press any key
to stop the server.", portno);

    Console.ReadLine();
}
```

Run the application. You will see a console screen with a message on it. Open a browser and type the following URL:

http://localhost:8888/

It will display the EchoService. Now change the URL to: http://localhost:8888/echo/DotNetCurry
You will see a hello message on the browser.

Conclusion

As we saw, ServiceStack is an awesome stack of technologies that makes it easier to build and secure an end-to-end application and host it on any platform depending on infrastructure provided by the enterprise. The technology has a number of other features including Logging, Caching, Filters, Validation and support for message queues. Check the official wiki pages for more details on these topics ■

Download the entire source code from our GitHub Repository at bit.ly/dncm16-servicestack

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

Authoring your First jQuery Plugin

One of the key aspects of jQuery is the possibility to extend it. A jQuery plugin is a method that *extends* the jQuery object. You can create a simple plugin by adding a method to $.fn in the following manner:

```
$.fn.plugin = function() {
}
```

If you look at the jQuery documentation, line 36 (https://github.com/jquery/jquery/blob/1.11.1/src/core.js#L36) you will find that the *fn* property is an alias to the *prototype* property, which returns the current jQuery version.

```
36  jQuery.fn = jQuery.prototype = {
37          // The current version of jQuery being used
38          jquery: version,
```

The jQuery Boilerplate plugin (https://github.com/jquery-boilerplate/jquery-boilerplate/blob/91b4eb05b18f3038a6f8a34c41435c4d2702c0d0/dist/jquery.boilerplate.js) by Zeno Rocha and Addy Osmani is a good starting point to start creating your own plugin. Here's the structure of the plugin:

```
/*
 *  jQuery Boilerplate - v3.3.3
 *  A jump-start for jQuery plugins development.
 *  http://jqueryboilerplate.com
 *
 *  Made by Zeno Rocha
 *  Under MIT License
 */; (function ($, window, document, undefined) {

  var pluginName = "defaultPluginName",
  defaults = {
    propertyName: "value"
  };

  function Plugin(element, options) {
    this.element = element;
    this.settings = $.extend({},
    defaults, options);
    this._defaults = defaults;
    this._name = pluginName;
    this.init();
  }

  $.extend(Plugin.prototype, {
    init: function () {
      console.log("xD");
    },
    yourOtherFunction: function () {
      // some logic
    }
  });

  $.fn[pluginName] = function (options){
    this.each(function () {
      if (!$.data(this, "plugin_" +
      pluginName)) {
        $.data(this, "plugin_" +
        pluginName, new Plugin(this,
        options));
      }
    });

    return this;
  };

})(jQuery, window, document);
```

Although the boilerplate plugin comes with some useful inline comments (not included in the above code) which you must read, I will try to simplify things for you by walking through this structure by creating our own plugin based on this template.

Create a new file 'SimplePlugin.html'. We will create a simple *highlight* plugin which can highlight a textbox or a div by giving it a background color and a border. This plugin is saved as 'jquery.highlighttextbox.js' in the scripts folder of the source code that comes with this article.

Once it is ready, we will be calling our plugin in the following manner:

```
<script type="text/javascript">
    $(function () {
        $("#myText").highlight();
    });
</script>
```

The first step is to understand the following structure of the boilerplate plugin:

```
;(function ($, window, document,
undefined) {
...

})(jQuery, window, document);
```

What you see here is an I*mmediately Invoked Function Expression a.k.a IIFE*. An IIFE is a function that is immediately defined and then invoked. Since JavaScript has a function scope, IIFE lets you create a 'private scope' which is safe from outside tampering. This separates your variables and other code from the global scope.

To make it simpler, think of it as an anonymous function wrapped within a function call. The function is invoked using the closing brackets ()

```
;(function() {
// Function body
})();
```

Also observe there is a semi-colon (;) before function invocation. The semi-colon is a safety net against concatenated scripts and/or other plugins which may not be closed properly.

Let us now briefly talk about the arguments (`$, window, document, undefined`).

'$' here simply represents a reference to a jQuery object that is passed in to the wrapper function and avoids plugin conflicts with other languages like Prototype, that also use $.

window, document and *undefined* are passed as local variables to the function. Passing these as arguments, shorten the time it takes to resolve these variables. However in my personal opinion, the time saved by using these local variables is negligible. So if your plugin does not reference *window* or *document* too many times in your code, you can skip passing these variables. For knowledge purpose, another reason *undefined* is passed is because the *undefined* global property in ECMAScript 3, is mutable; so that means someone could change its value from outside. However in ECMAScript 5, *undefined* has been made immutable (read only).

To access the actual *window* and *document* objects, you are passing (`jQuery, window, document`) parameters at the end of your anonymous function.

Our next step is to define the plugin name and create defaults for it.

```
;(function ($, window, document,
undefined) {

  var pluginName = "highlight",
  // plugin defaults:
  defaults = {
    bgColor: "AliceBlue",
    border: "3px solid #cdcdcd"
  };
```

Our plugin will be called 'highlight'. *defaults* allows users to pass options to the plugin. You can even send a single argument as an object literal, which is a standard jQuery practice. It is also a best practice to set default options using the defaults object, hence we are setting the default background color and border in case the user does not pass any option.

The next step is to define the plugin constructor which does all the heavy lifting and creates a few instance properties and calls init().

```
function Plugin(element, options) {
  this.element = element;
  this.settings = $.extend({}, defaults,
  options);
  this._defaults = defaults;
  this._name = pluginName;
  this.init();
}
```

Observe how the boilerplate stores the custom settings of the user in `this.settings`. This is done to ensure that the plugin's default settings aren't overwritten by the user's custom settings.

The next step is the `init()` function which is called during initialization and contains the logic associated with the plugin initialization.

```
$.extend(Plugin.prototype, {
  init: function () {
    $(this.element).css({
      'background-color': this.settings.
      bgColor,
      'border': this.settings.border
    });
  }
});
```

To access the default or custom settings, we simply need to access `this.settings`.

The final step is to define a really lightweight plugin wrapper around the constructor, which prevents against multiple instantiations.

```
$.fn[pluginName] = function (options) {
  this.each(function () {
    if (!$.data(this, "plugin_" +
    pluginName)) {
      $.data(this, "plugin_" +
      pluginName, new Plugin(this,
      options));
    }
  });

  // chainable jQuery functions
  return this;
};
```

Within the function, *this* represents the jQuery object on which your function was called. That means that when you say `$("#myText").highlight()`, the value of *this* refers to the jQuery object containing the result of `$("#myText")`. Here you are referring to just one textbox but you could always do `$('div').highlight()` which will contain all the divs on the page. By doing `this.each`, you are looping over every element within *this*, and thus highlighting every div/text on the page.

Within `this.each`, we check to see if the textbox object has been assigned a plugin. If yes, that means the plugin is already up and running. In case we haven't, then we create a new Plugin() object and add it to the textbox element.

At the end of it, *this* is returned making your plug-in chainable. This means someone could do the following:

```
$("#myText").highlight().fadeOut();
```

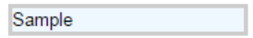
That's it. Your plugin is ready to be used. Define a textbox on the page:

```
<input type="text" id="myText"
name="myText" value="Sample" />
```

..and call our plugin in the following manner:

```
<script type="text/javascript">
  $(function () {
    $("#myText").highlight();
  });
</script>
```

You will see that the textbox gets a background color and a border:



Note: *An additional point I want to make here is that there are some jQuery plugins that act as Singletons. They define themselves directly on $ and do not return this. Eg:* `$.somePlugin = function()`. *Check the jQuery Cookie plugin for an example.*

Summarizing the pattern, some points to keep in mind are as follows:

- Always wrap your plugin in a self-invoking function
- Add a semi-colon before the function's invocation
- Use a basic defaults object and extend the options with defaults
- Use a simple plugin constructor for defining the logic for initial creation and assignment of the DOM element to work with
- always return the *this* keyword to maintain chainability unless you want to create a Singleton plugin
- Use a lightweight wrapper around the constructor, which helps to avoid issues such as multiple instantiations.

With the jQuery Boilerplate Plugin and these simple guidelines in mind, we can build simple to very complex and structured plugins in no time.

Further Reading: http://api.jquery.com/jQuery.extend/ ■

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

Getting Started with HTML5 Web Components

Image Courtesy: webcomponents.org

Web Components (WC) are a new set of web platform features that enable developers to build applications in a declarative, composable way.
~ webcomponent.org

Web development can be confusing and frustrating at times. Let's consider the following scenario – you want to create a web page with some common components such as a navigation bar or a breadcrumb. A quick way to do this is to grab some boilerplate templates, for example using the Twitter Bootstrap library, or create the components on your own using a JavaScript library, for example jQuery. HTML does not support such commonly used components. HTML supports some new semantic elements that were added as part of HTML5 specifications. However HTML wasn't extensive enough to create our own reusable components.

This situation has changed. In the past few months, we are hearing more and more about a new standard called **Web Components**. A lot of known web development experts claim that it will revolutionize the way we are creating web apps. Some of the major browser vendors are working on the Web Components polyfill libraries that can help bridge the gap between current browsers and the emerging standard. Will the standard revolutionize the way we build web apps? It definitely will, but until this standard is included in every browser, we will have to use a polyfill library.

In the article, you will get an overview of Web Components. You will get familiar with the four HTML5 APIs that enables you to create your own components and you will see a simple component implementation. At the end we will discuss the browser support and briefly discuss options to bridge the gap in browsers that do not support Web Components. But first things first, let us start by understanding what is the web components standard.

What are Web Components?

When you write traditional web applications which use server rendering engines for building web pages, you probably reuse components using custom/user controls. For example, one of the first tasks that I implemented as a novice ASP.Net developer was to create a drop down user control that was reused across an entire web site. In today's modern web development, the rendering process is moving to the front-end and developers want to use the same custom control techniques when building the front-end. But how can a developer create his own reusable DOM element? Can we do that using HTML standards that are in practice today?

Web Components is a bundle of new standards that allow developers to build their own DOM elements. The custom elements can encapsulate common behavior and element interaction endpoints, such as custom events. A web component can also expose custom attributes that can be later used to customize the element and for setting the element's behavior.

A custom web component is used in a declarative way like any other HTML element. The custom elements that you create actually denote something meaningful, for example a navbar element; instead of a generic or elements which expresses nothing. Once you go the web component way, your HTML will start to be meaningful and more robust.

Note - *If you are using AngularJS, the concept of web components might sound familiar. This is because AngularJS directives use some of the concepts written in the web components standard.*

The Web Components standard includes four different parts:

1. Templates
2. Imports
3. Shadow DOM
4. Custom Elements

In the next sections, we will explore each of these four parts that are included in the standard.

Templates

In the past when you wanted to create a template, you had two main options. The first option was to use a block element that would be hidden initially and displayed later when some web page interaction occurred. The problem with this option is that you fetch resources from the server, for example images, even though they might not be

shown to the user. The second option was to create a script tag with an id and give it the text/template type. In that script tag, you would plant your DOM code. For example, here is how you might have created a template:

```
<script id=”myTemplate” type=”text/
template”>
  <div>
    …
  </div>
</script>
```

In this option the web page won't load unnecessary resources but in order to use the template, you need to fetch the script tag element and extract its HTML. That might lead to cross-site scripting vulnerability. We can do better than that.

The new *template* element, which was introduced as part of the Web Components standard, enables you to define a HTML blueprint that can be later used to instantiate document fragments which include the same structure. Now you have the ability to wrap HTML content including script and style tags inside a template element and later on to use the template in your code. Here is an example of the previous script tag but now defined as a template:

```
<template id=”myTemplate”>
  <div>
    …
  </div>
</template>
```

In order to use the template you will have to use JavaScript. The following lines of code will fetch the template element and will import a cloned node. Later on, you would be able to append the clone into the DOM.

```
var template = document.
querySelector(‘#myTemplate’);
var clone = document.
importNode(template.content, true);
```

Once the clone is appended, if the template includes script tags or style tags, they will be executed. If the template includes resources such as images or videos, they will be retrieved from the server.

One last thing that you must know about the template element is that it doesn't support data binding. The template element as expected is a template and nothing more. We will discuss how to do data binding later on in the article.

Note – you can find the full templates documentation here: http://www.w3.org/TR/html5/scripting-1.html#the-template-element

Imports

The new HTML5 Imports specification enables developers to load additional HTML documents without the need to use Ajax. Imports are new type of link tag. If you are a web developer you probably used the rel=”stylesheet” to specify that a link element should load a CSS file. In imports, you replace the *stylesheet* string with import and that's it, the link now specifies an *import*. Here is an example:

```
<link rel=”import” href=”myImport.html”>
```

In the example you can see that we are loading a HTML file called myImport.

Imports enables web developers to bundle a full component into a HTML file which might include scripts and CSS styles and later on to import that bundle in a single call. For example, if you have a component that includes JavaScript files, CSS files and more, you can bundle them into a HTML document and just import it. This option makes component writing much more powerful.

Importing a document using the import link doesn't include its content into the web page. What it really does is to grab the document and parse it and of course load all the document's additional resources. After the document is loaded and parsed, it is available through JavaScript. In order to access the content of an import, you need to use the link element import property. The following example shows you how to get the content from an import link element:

```
var content = document.
querySelector(‘link[rel=”import”]’).
import;
```

In the example, the content variable will include the whole content in memory document that the import retrieved. Once you have a reference to the document, you can use JavaScript and use its content. Let's take a look at an example that uses both a template element and HTML5 import. Here is the template code that exists in myTemplate.html file:

```
<!-- This template element exists in
myTemplate.html file -->
<template id=”myTemplate”>
  <h1>My template rules!</h1>
  <script>console.log(“Log executed once
the template is activated”);</script>
</template>
```

Here is the document that will import the template:

```
<html>
<head>
  <link rel=”import” href=” myTemplate.
html”>
</head>
<body>
  <div id=”shell”></div>
  <script>
    var importContent = document.
    querySelector(‘link
    [rel=”import”]’).import;

    var template = importContent.
    querySelector(‘#myTemplate’);
    var clone = document.
    importNode(template.content, true);

    document.querySelector(‘#shell’).
    appendChild(clone);
  </script>
</body>
</html>
```

In the previous code example, once the import retrieves the HTML document that includes the template, we select the element. Later on, we import a clone of the template and append it into a DIV element with a shell id. Now that we are familiar with templates and import, the next stop in the Web Components standard is the shadow DOM.

Shadow DOM

The shadow DOM specification describes how to encapsulate DOM parts such that the browser will know how to present those parts but won't show them in the source code. That creates a boundary between the component and its user.
The concept of shadow DOM isn't new and it is widely used by browser vendors to hide implementation details of already made components. For example, have you ever wondered how you get the control buttons of a video tag? You guessed right, this is shadow DOM in practice. Just open Chrome Developer Tools by pressing F12, go to settings and in the General tab enable the *Show user agent shadow DOM*. The following figure shows the configuration option:

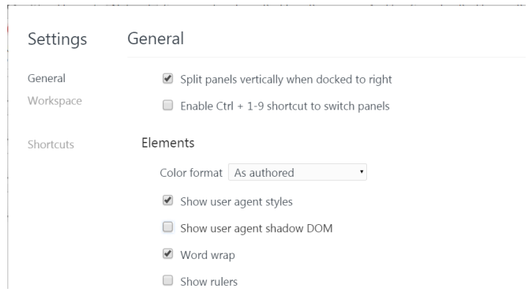


Figure 1: Enabling Chrome to Show the Shadow DOM

Once you set the configuration option, go to any web page that includes video or audio control and look at its real implementation. Here is an example of a Textarea element once you enable the shadow DOM in the browser:

```
▼<textarea name="csi" id="csi" style="display:none">
  ▼#shadow-root (user-agent)
    <div id="inner-editor">1</div>
  </textarea>
```

Figure 2: Textarea Element with its Shadow Root

In order to create a shadow DOM, you will have to select an HTML element and call the createShadowRoot function on it. The returned value will be a document fragment that you will be able to populate and fill with your desired content. The following source code shows how to wrap a DIV with some content inside a shadow DOM host

element:

```
<div id=”shadowDOMHost”></div>

<script>
  var host = document.
  querySelector(‘#shadowDOMHost’);
  var root = host.createShadowRoot();
  root.innerHTML = ‘<div>Lurking in the
shadows</div>’
</script>
```

Once you created the shadow DOM root, a shadow boundary encapsulates all the root’s inner implementation. The shadow boundary prevents outer CSS to affect the inner content. More than that, it also prevents external JavaScript code to reach the wrapped content. This behavior guards the implementations from the outside world and is really useful when you want to create custom web components that aren’t affected by the hosting web page. Now that you are familiar with shadow DOM, it is time to explore the last Web Components part – custom elements.

Custom Elements

Web page development includes a lot of HTML. If you will open some web page HTML source, you will probably see large amount of DIV elements. Divs are used as block elements that wrap some content but they don’t have semantic. Instead of abusing the DIV element and using it everywhere, wouldn’t it be nicer to create an element that expresses its real meaning? This is where the custom elements specification comes in handy.

Custom elements specification enables web developers to extend HTML and to create their own custom **HTML elements**. The only restriction is that the new element name should include a dash and its prototype must extend the HTMLElement object. In order to create a new element, you need to register it. You will need to use the d**ocument. registerElement** function which receives a mandatory name argument and an optional object that describes the new element’s prototype. If you omit the second argument, by default the custom element will inherit from **HTMLElement**. The following code shows how to register an element called my-element:

```
var myElement = document.
registerElement(‘my-element’);
```

The call to registerElement will explain to the browser what is the new element and return a constructor function that enables to instantiate instances of the new element.

Custom elements also enable to extend native HTML elements. The following code shows how to extend an input type:

```
var myInput = document.
registerElement(‘my-input’, {
  prototype: Object.
  create(HTMLInputElement.prototype),
  extends: ‘input’
});
```

As you can see from the example, we pass a configuration option and state that the new element prototype is going to be the **HTMLInputElement** prototype. We also state that the element extends the input element. Doing that ensures that our new element extends the native one.

Once you created your new custom element, the way to use it is very simple. Just put in the HTML , a tag with the name that you gave to your custom element and you are done. The following HTML shows how to add my-input in your HTML:

```
<my-input></my-input>
```

Another option to create a new instance of the new custom element is by using the regular **document. createElement** function. Pay attention that you must register the element before you use the **createElement** function:

```
var myInput = document.
registerElement(‘my-input’, {
  prototype: Object.
  create(HTMLInputElement.prototype),
  extends: ‘input’
});
var myInputInstance = document.
createElement(‘my-input’);
document.body.
appendChild(myInputInstance);
```

Right now we only extended an element but we didn’t add any new behavior to the element we created. You can add new properties and functions to the custom element you created by decorating the prototype that you pass to the registerElement function. The following code shows one example of adding new property and function to the prototype of the new registered element:

```
var XFoo = document.registerElement(‘x-
foo’, {
  prototype: Object.
  create(HTMLInputElement.prototype, {
    myProperty: {
      get: function() { return
      ‘myProperty’; }
    },
    myFunction: {
      value: function() {
        console.log(‘Doing something’);
      }
    }
  }),
  extend: ‘input’
});
```

In a new custom element, you can attach event handlers to a few events that are triggered during the life time of the element. For example, if you want to do something when an instance of the element is created, you can use the **createdCallback** event handler. Another useful event is the **attributeChangedCallback** that enables you to wire an event and listen for changes in attributes. The **attributeChangedCallback** event can be used in data binding scenarios where you want to listen to changes in attributes and act accordingly. The following example shows how to register the **createdCallback** function:

```
var myPrototype = Object.
create(HTMLInputElement.prototype);
myPrototype. createdCallback =
function() {
  // Do something when an new instance
  of the element is created
};

var myElement = document.
registerElement(‘my-element’, {
prototype: myPrototype });
```

Now that you are familiar with all the different parts of the Web Components specifications, we can start and building our new components.

Web Component in Action

The following example shows how to create a simple Web Component that will display an image and a caption for a team member. We will start with the code of the web component which resides in the tmpl.html file:

```
<template id=”teamMemberTmpl”>
 <style>
  .teamMember {
   float: left;
   width: 152px;
   height: 285px;
   border: 1px solid #262262;
   background-color: #279bd4;
   margin-bottom: 20px;
  }
  .teamMember > .photo {
   height: 225px;
   width: 150px;
   background-color: #fff;
   text-align: center;
  }
  .teamMember > .name {
   color: #fff;
   font-size: 15px;
   width: 150px;
   text-align: center;
  }
  .teamMember > .title {
   color: #fff;
   font-size: 12px;
   width: 150px;
   text-align: center;
  }
 </style>
 <div title=”Gil Fink”
 class=”teamMember”>
  <div class=”photo”>
   <img alt=”” src=”” width=”150”
   height=”225”>
  </div>
  <div class=”name”> </div>
   <div class=”title”></div>
 </div>
</template>
<script>
 var thisDoc = document.currentScript.
 ownerDocument;
 var teamMemberTmpl = thisDoc.
 querySelector(‘#teamMemberTmpl’);

// Create a prototype for a new element
 that extends HTMLElement
 var teamMemberPrototype = Object.
```

```
create(HTMLElement.prototype);
// Setup our Shadow DOM and clone the
template
teamMemberPrototype.createdCallback =
function () {
 var root = this.createShadowRoot();
 root.appendChild(document.
 importNode(teamMemberTmpl.content,
 true));

 var img = root.querySelector(“.
 teamMember > .photo img”);
 img.setAttribute(‘src’, this.
 getAttribute(‘data-src’));
 img.setAttribute(‘alt’, this.
 getAttribute(‘data-alt’));

 var name = root.querySelector(‘.
 name’);
 name.textContent = this.
 getAttribute(‘data-name’);

 var title = root.querySelector(‘.
 title’);
 title.textContent = this.
 getAttribute(‘data-title’);
};
// Register our new element
var teamMember = document.
registerElement(‘team-member’, {
 prototype: teamMemberPrototype
 });
</script>
```

We start by defining the template that is going to create each team member. The template includes the HTML code and the styles that will decorate the HTML. After defining how the template looks like, we add a script that will define the custom element. Since the web component will be imported from another web page, we use the first line to get the owner document. From the owner document, we can extract the template and use it to create the shadow DOM and attach the DOM element to the hosting web page. In the createdCallback function of the web component, we set some properties to the image and the DIV elements in the template.

Now that we have our web component set, we can start importing it into our web pages. The following code shows a web page that hosts two team member elements:

```
<html>
<head>
  <link rel=”import” href=”tmpl.html”>
</head>
<body>
  <div id=”shell”>
    <team-member data-alt=”Gil Fink
    Image” data-name=”Gil Fink” data-
    title=”Senior Consultant” data-
    src=”http://sparxys.azurewebsites.
    net/Content/Images/Team/GilFink.
    jpg”></team-member>
    <team-member data-alt=”Another
    Image” data-name=”Gil Fink1”
    data-title=”Senior Consultant” data-
    src=”http://gilfink.azurewebsites.
    net/Images/Carousel/Gil2.jpg”></
    team-member>
  </div>
</body>
</html>
```

Running the code will result in the following output:

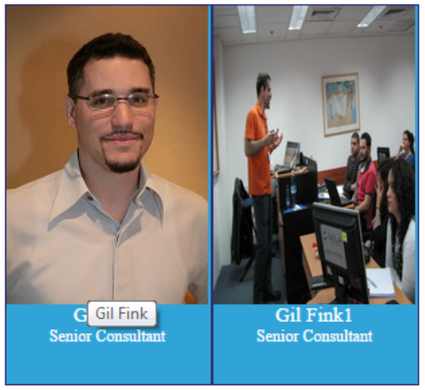


Figure 3: The Result of Running the Web Page

You will be able to run this code successfully only in web components supporting browsers and that brings us to the next article section.

Web Components Polyfill Libraries

One of the larger problems that you will encounter when you want to start using Web Components is browser support. Most of the APIs that we discussed in the article aren’t supported in most of the major browsers. For example at the time of writing the article, Internet Explorer doesn’t support all Web Components APIs in none of its versions (even Internet Explorer 11). Firefox supports templates and imports but doesn’t support shadow DOM or custom elements. Chrome and Opera support all the APIs.

If you want to use tomorrow’s standards today, you should consider using a *polyfill* library. A polyfill library is a JavaScript library that mimics the expected browser functionality, if it doesn’t exists. The major libraries that you can use are Polymer and X-Tag. Both of the libraries include ways to create custom web components. More than that, both of the libraries come with a ready to use custom components set. In Polymer, the ready to use components are part of the library. In X-Tag you can use the Brick library.

In this article, we won’t cover these libraries as they deserve a complete article of its own.

Summary

Web components standard offers a really revolutionary direction for building web applications front-end in Lego style. You can divide your web page into reusable components which include their own behavior and interaction. Later on, you can composite your page using the components that you’ve created. This is a really important direction in the way we think about how to build web applications. Even though the standard is still in development, libraries such as X-Tag and Polymer can help you get started and build your future web applications ■

Download the entire source code from our GitHub Repository at bit.ly/dncm16-html5webc

About the Author

Gil Fink is a web development expert and ASP.Net/IIS Micrsoft MVP. He is the founder and owner of sparXys, a consulting company. He is currently consulting for various enterprises and companies, where he helps to develop Web and RIA-based solutions. He conducts lectures and workshops for individuals and enterprises who want to specialize in infrastructure and web development. He is also co-author of several Microsoft Official Courses (MOCs) and training kits, co-author of “Pro Single Page Application Development” book (Apress) and the founder of Front-End.IL meetup. You can learn more about Gil on his website http://www.gilfink.net