ISSUE 15 | NOV-DEC 2014 EDITION

DNCMagazine

www.dotnetcurry.com

SOFTWARE GARDENING: HARVESTING

Performance Optimization in ASP.NET Web Sites

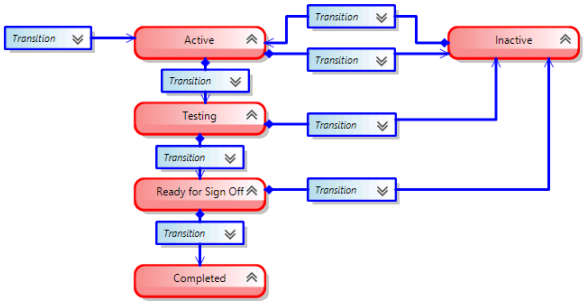
AZURE Blob Storage Snapshots

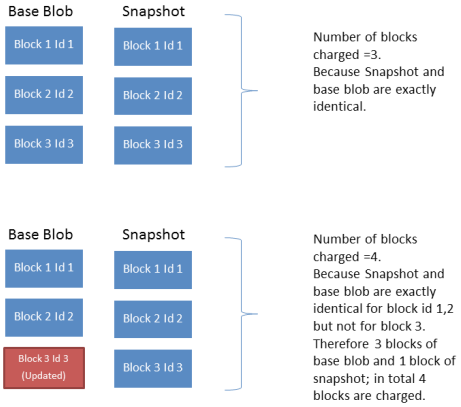
51 AngularJS Tips

Introduction to ServiceStack

Customizing Test Plan and Test Suite Work Items using MTM 2013

CONTENTS





EDITORIAL

Welcome to the fifteenth edition of the DNC Magazine. We wish our friends from India a very Happy Diwali!

This edition warms up with an awesome AngularJS Cheatsheet compiled by Ravi and me. If you are working on an Angular project, make sure you take a printout and pin it to your soft board. Craig Berntson in his Software Gardening column, follows up on the previous edition's column, debunking some myths of comparing software to construction. In this edition, Craig stresses on why Continuous Integration is so important to deliver working software.

Raj Aththanayake talks about how to optimize performance of ASP.NET websites in the browser and at server-side. Gouri talks about how we can now customize Test Plan and Test Suite artefact's in the latest Visual Studio 2013 Update 3. We also have an interesting new section this time where Ravi Kiran shows us how to use an alternative stack of technologies called ServiceStack to build web apps and APIs.

We are happy to welcome new contributing writer Kunal Chandratre who shows how to perform various operations on Azure Blob Storage Snapshots using REST and Client Library. Finally I wrap up this edition with a simple introduction to a JavaScript Templating Library called Mustache.js.

So from the entire team of 'DNC Magazine', here's wishing all our readers a joyous Holiday Season! We'll see you again in the New Year!

Suprotim Agarwal
Editor in Chief

CREDITS

Editor In Chief Suprotim Agarwal
suprotimagarwal@dotnetcurry.com

Art Director Minal Agarwal

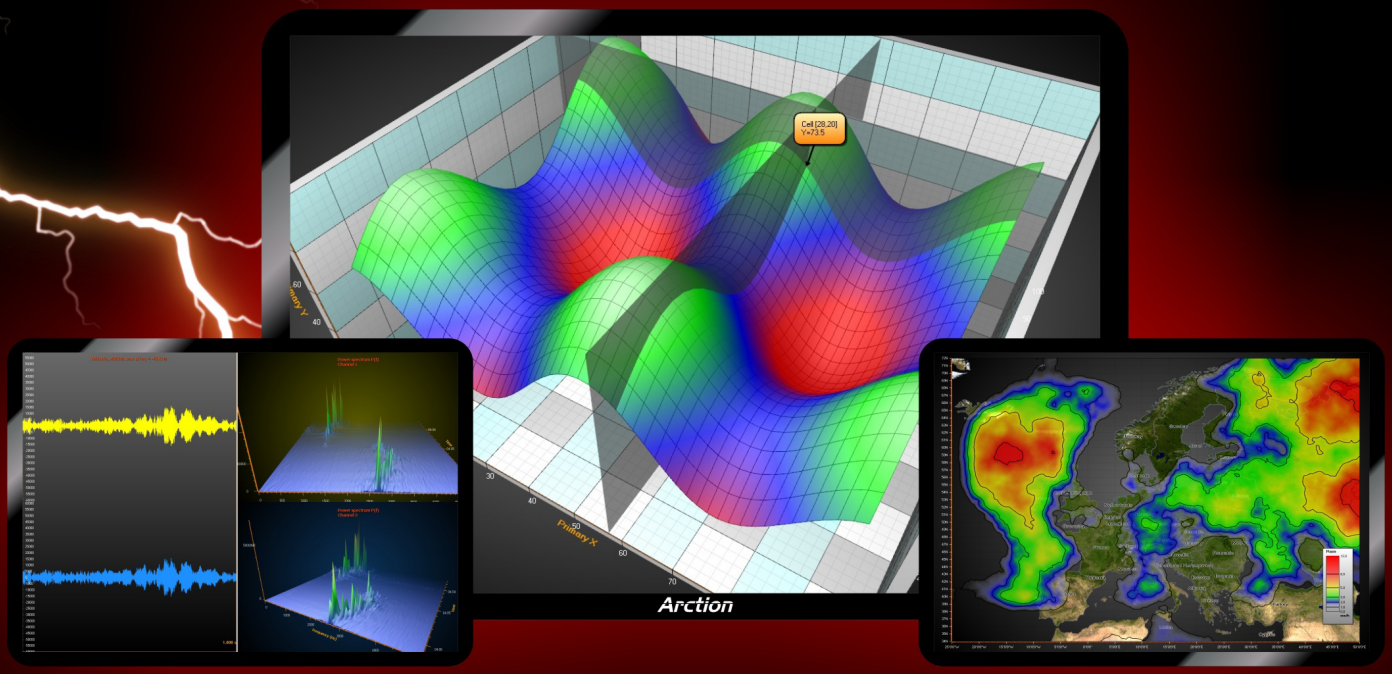
Contributing Writers Craig Berntson, Gouri Sohoni, Kunal Chandratre, Raj Aththanayake, Ravi Kiran, Suprotim Agarwal

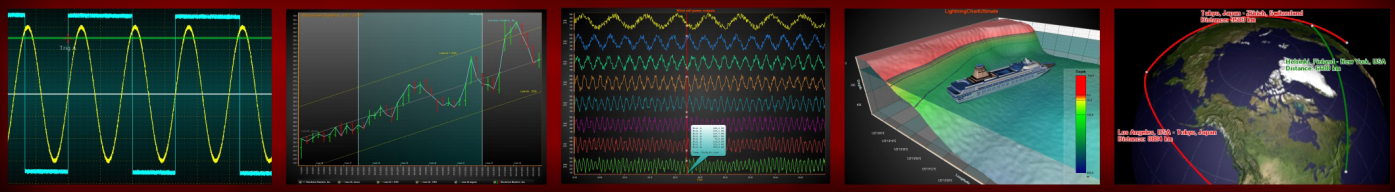
Reviewers: Alfredo Bardem, Subodh Sohoni, Ravi Kiran, Carol Nadarwalla, Suprotim Agarwal

Next Edition 2nd Jan 2015
www.dncmagazine.com

The fastest rendering data visualization components
for WPF and WinForms...

LightningChart





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

WPF charts performance comparison

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

Winforms charts performance comparison

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

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

- Entirely DirectX GPU accelerated
- Superior 2D and 3D rendering performance
- Optimized for real-time data monitoring
- Touch-enabled operations
- Supports gigantic data sets
- On-line and off-line maps
- Great customer support
- Compatible with Visual Studio 2005...2013

Download a free 30-day evaluation from

www.LightningChart.com

Arction

Pioneers of high-performance data visualization

AngularJS CHEAT SHEET

AngularJS is an extensible and exciting new JavaScript MVC framework developed by Google for building well-designed, structured and interactive single-page applications (SPA). It lays strong emphasis on Testing and Development best practices such as templating and declarative bi-directional data binding.

This cheat sheet co-authored by Ravi Kiran and Suprotim Agarwal, aims at providing a quick reference to the most commonly used features in AngularJS. It will also make you quickly productive with Angular.

The cheat sheet is based on Angular v2 and is divided into Beginners and Intermediate/Advanced Developer categories:

BEGINNER

01 Important AngularJS Components and their usage:

- angular.module() defines a module
- Module.controller() defines a controller
- Module.directive() defines a directive
- Module.filter() defines a filter
- Module.service() or Module.factory() or Module.provider() defines a service
- Module.value() defines a service from an existing object Module
- ng-app attribute sets the scope of a module
- ng-controller attribute applies a controller to the view
- $scope service passes data from controller to the view
- $filter service uses a filter
- ng-app attribute sets the scope of the module

02 Bootstrapping AngularJS application:

Bootstrapping in HTML:

```
<div ng-app="moduleName"></div>
```

Manual bootstrapping:

```
angular.bootstrap(document,["moduleName"])
```

03 Expressions:

`{{ 4+5 }}` -> yields 9

`{{ name }}` -> Binds value of name from current scope and watches for changes to name

`{{ ::name }}` -> Binds value of name from current scope and doesn't watch for change (Added in AngularJS 1.3)

04 Module:

Create a module named myModule1 that depends on myModule2 and myModule2:

```
angular.module("myModule1",["myModule2",
"myModule2"])
```

Get reference to the module myModule1

```
angular.module("myModule1")
```

05 Defining a Controller and using it:

i. With $scope:

```
angular.module("myModule").
controller("SampleController",
function($scope,){
  //Members to be used in view for binding
  $scope.city="Hyderabad";
});
```

In the view:

```
<div ng-controller="SampleController">
  <!-- Template of the view with binding
  expressions using members of $scope -->
  <div>{{city}}</div>
</div>
```

ii. Controller as syntax:

```
angular.module("myModule").
controller("SampleController", function(){
  var controllerObj = this;
  //Members to be used on view for binding
  controllerObj.city="Hyderabad";
});
```

In the view:

```
<div ng-controller="SampleController as ctrl">
  <div>{{ctrl.city}}</div>
</div>
```

06 Defining a Service:

```
angular.module("myModule").
service("sampleService", function(){
  var svc = this;
  var cities=["New Delhi", "Mumbai",
  "Kolkata", "Chennai"];
  svc.addCity = function(city){
    cities.push(city);
  };

  svc.getCities = function(){
    return cities;
  }
});
```

The members added to instance of the service are visible to the outside world. Others are private to the service. Services are singletons, i.e. only one instance of the service is created in the lifetime of an AngularJS application.

07 Factory:

```
angular.module("myModule").
factory("sampleFactory", function(){
  var cities = ["New Delhi", "Mumbai",
  "Kolkata", "Chennai"];
  function addCity(city){cities.push(city);}
  function getCities(){return cities;}
  return{
    getCities: getCities,
    addCity:addCity
  };
});
```

A factory is a function that returns an object. The members that are not added to the returning object, remain private to the factory. The factory function is executed once and the result is stored. Whenever an application asks for a factory, the application returns the same object. This behavior makes the factory a singleton.

08 Value:

```
angular.module("myModule").value("sampleValue", {
  cities : ["New Delhi", "Mumbai", "Kolkata",
  "Chennai"],
  addCity: function(city){cities.push(city);},
  getCities: function(){return cities;}
});
```

A value is a simple JavaScript object. It is created just once, so value is also a singleton. Values can't contain private members. All members of a value are public.

09 Constant:

```
angular.module("myModule").
constant("sampleConstant",{pi: Math.PI});
```

A constant is also like a value. The difference is, a constant can be injected into config blocks, but a value cannot be injected.

10 Provider:

```
angular.module("myModule").
provider("samplePrd", function(){
  this.initCities = function(){
    console.log("Initializing Cities…");
  };

  this.$get = function(){
    var cities = ["New Delhi", "Mumbai",
    "Kolkata", "Chennai"];
    function addCity(city){
      cities.push(city);
    }
    function getCities(){return cities;}
      return{ getCities: getCities,addCity:addCity
    };
  }
});
```

A provider is a low level recipe. The $get() method of the provider is registered as a factory. Providers are available to config blocks and other providers. Once application configuration phase is completed, access to providers is prevented.

After the configuration phase, the $get() method of the providers are executed and they are available as factories. Services, Factories and values are wrapped inside provider with $get() method returning the actual logic implemented inside the provider.

11 Config block:

```
angular.module("myModule").config(function
(samplePrdProvider, sampleConstant){
  samplePrdProvider.init();
  console.log(sampleConstant.pi);
});
```

Config block runs as soon as a module is loaded. As the name itself suggests, the config block is used to configure the application. Services, Factories and values are not available for config block as they are not created by this time. Only providers and constants are accessible inside the config block. Config block is executed only once in the lifetime of an Angular application.

12 Run block:

```
angular.module("myModule").run(function(<any
services, factories>){
  console.log("Application is configured. Now inside run
  block");
});
```

Run block is used to initialize certain values for further use, register global events and anything that needs to run at the beginning of the application. Run block is executed after config block, and it gets access to services, values and factories. Run block is executed only once in the lifetime of an Angular application.

13 Filters:

```
angular.module("myModule").
filter("dollarToRupeee",  function(){
  return function(val){
    return "Rs. " + val * 60;
  };
});
```

Usage:

```
<span>{{price | dollarToRupee}}</span>
```

Filters are used to extend the behavior of binding expressions and directives. In general, they are used to format values or to apply certain conditions. They are executed whenever the value bound in the binding expression is updated.

14 Directives:

```
myModule.directive("directiveName", function
(injectables) {
  return {
    restrict: "A",
    template: "<div></div>",
    templateUrl: "directive.html",
    replace: false,
    transclude: false,
    scope: false,
```

```
      require: ["someOtherDirective"],
      controller: function($scope, $element,
      $attrs, $transclude, otherInjectables) { ... },
      link: function postLink(scope, iElement,
     iAttrs) { ... },
      priority: 0,
      terminal: false,
      compile: function compile(tElement, tAttrs,
     transclude) {
      return {
         pre: function preLink(scope, iElement,
         iAttrs, controller) { ... },
         post: function postLink(scope,
         iElement, iAttrs, controller) { ... }
      }
     }
   };
});
```

Directives add the capability of extending HTML. They are the most complex and the most important part of AngularJS. A directive is a function that returns a special object, generally termed as *Directive Definition Object*. The Directive Definition Object is composed of several options as shown in the above snippet. Following is a brief note on them:

- **restrict:** Used to specify how a directive can be used. Possible values are: E (element), A (Attribute), C (Class) and M (Comment). Default value is A

- **template:** HTML template to be rendered in the directive

- **templateUrl:** URL of the file containing HTML template of the element

- **replace:** Boolean value denoting if the directive element is to be replaced by the template. Default value is false

- **transclude:** Boolean value that says if the directive should preserve the HTML specified inside directive element after rendering. Default is false

- **scope:** Scope of the directive. It may be same as the scope of surrounding element (default or when set to false), inherited from scope of the surrounding element (set to true) or an isolated scope (set to {})

- **require:** A list of directive that the current directive needs. Current directive gets access to controller of the required directive. An object of the controller is passed into link function of the current directive.

- **controller:** Controller for the directive. Can be used to manipulate values on scope or as an API for the current directive or a directive requiring the current directive

- **priority:** Sets priority of a directive. Default value is 0. Directive with higher priority value is executed before a directive with lower priority

- **terminal:** Used with priority. If set to true, it stops execution of directives with lower priority. Default is false

- **link:** A function that contains core logic of the directive. It is executed after the directive is compiled. Gets access to scope, element on which the directive is applied (jqLite object), attributes of the element containing the directive and controller object. Generally used to perform DOM manipulation and handling events

- **compile:** A function that runs before the directive is compiled. Doesn't have access to scope as the scope is not created yet. Gets an object of the element and attributes. Used to perform DOM of the directive before the templates are compiled and before the directive is transcluded. It returns an object with two link functions:

  - **pre link:** Similar to the link function, but it is executed before the directive is compiled. By this time, transclusion is applied

  - **post link:** Same as link function mentioned above

15 Most used built-in directives:

- **ng-app:** To bootstrap the application

- **ng-controller:** To set a controller on a view

- **ng-view:** Indicates the portion of the page to be updated when route changes

- **ng-show / ng-hide:** Shows/hides the content within the directive based on boolean equivalent of value assigned

- **ng-if:** Places or removes the DOM elements under

this directive based on boolean equivalent of value assigned

- **ng-model:** Enables two-way data binding on any input controls and sends validity of data in the input control to the enclosing form

- **ng-class:** Provides an option to assign value of a model to CSS, conditionally apply styles and use multiple models for CSS declaratively

- **ng-repeat:** Loops through a list of items and copies the HTML for every record in the collection

- **ng-options:** Used with HTML select element to render options based on data in a collection

- **ng-href:** Assigns a model as hyperlink to an anchor element

- **ng-src:** Assigns a model to source of an image element

- **ng-click:** To handle click event on any element

- **ng-change:** Requires ng-model to be present along with it. Calls the event handler or evaluates the assigned expression when there is a change to value of the model

- **ng-form:** Works same as HTML form and allows nesting of forms

- **ng-non-bindable:** Prevents AngularJS from compiling or binding the contents of the current DOM element

- **ng-repeat-start** and **ng-repeat-end:** Repeats top-level attributes

- **ng-include:** Loads a partial view

- **ng-init:** Used to evaluate an expression in the current scope

- **ng-switch** conditionally displays elements

- **ng-cloak** to prevent Angular HTML to load before bindings are applied.

16 AngularJS Naming Conventions

- While naming a file say an authentication *controller*, end it with the object suffix. For eg: an authentication controller can be renamed as auth–controller.js. Similar *service* can be called as auth-service.js, *directive* as auth-directive.js and a *filter* as auth-filter.js

- Create meaningful & short lower case file names that also reflect the folder structure. For eg: if we have a login controller inside the login folder which is used for creating users, call it login-create-controller.js

- Similar a testing naming convention that you could follow is if the filename is named as login-directive.js, call its test file counterpart as login-directive_test.js. Similarly a test file for login-service.js can be called as login-service_test.js
Use a workflow management tool like *Yeoman* plugin for Angular that automates a lot of these routines and much more for you. Also look at *ng-boilerplate* to get an idea of the project and directory structure.

17 Dependency Injection:

AngularJS has a built-in dependency injector that keeps track of all components (services, values, etc.) and returns instances of needed components using dependency injection. Angular's dependency injector works based on names of the components.

A simple case of dependency injection:

```
myModule.controller("MyController", function($scope,
$window, myService){
});
```

Here, $scope, $window and myService are passed into the controller through dependency injection. But the above code will break when the code is minified. Following approach solves it:

```
myModule.controller("MyController", ["$scope",
"$window", "myService",
        function($scope, $window, myService){
}]);
```

18 Routes

Routes in AngularJS application are defined using $routeProvider. We can define a list of routes and set one of the routes as default using otherwise() method; this route will respond when the URL pattern doesn't match any of the configured patterns.

19 Registering routes:

```
myModule.config(function($routeProvider){
  $routeProvider.when("/home",
  {templateUrl:"templates/home.html",
  controller: "HomeController"})
  .when("/details/:id", {template:
  "templates/details.html",
   controller:"ListController"})
  .otherwise({redirectTo: "/home"});
});
```

20 Registering services:

Angular provides us three ways to create and register our own Services – using Factory, Service, and Provider. They are all used for the same purpose. Here's the syntax for all the three:

```
Service: module.service( 'serviceName', function );
Factory: module.factory( 'factoryName', function );
Provider: module.provider( 'providerName', function );
```

The basic difference between a service and a factory is that service uses the constructor function instead of returning a factory function. This is similar to using the *new* operator. So you add properties to 'this' and the service returns 'this'.

With Factories, you create an object, add properties to it and then return the same object. This is the most common way of creating Services.

If you want to create module-wide configurable services which can be configured before being injected inside other components, use Provider. The provider uses the $get function to expose its behavior and is made available via dependency injection.

21 Some useful utility functions

- **angular.copy -** Creates a deep copy of source
- **angular.extend -** Copy methods and properties from one object to another
- **angular.element -** Wraps a raw DOM element or HTML string as a jQuery element
- **angular.equals -** Determines if two objects or two values are equivalent
- **angular.forEach -** Enumerate the content of a collection
- **angular.toJson -** Serializes input into a JSON-formatted string
- **angular.fromJson -** Deserializes a JSON string
- **angular.identity -** Returns its first argument
- **angular.isArray -** Determines if a reference is an Array
- **angular.isDate -** Determines if a value is a date
- **angular.isDefined -** Determines if a reference is defined
- **angular.isElement -** Determines if a reference is a DOM element
- **angular.isFunction -** Determines if a reference is a Function
- **angular.isNumber -** Determines if a reference is a Number
- **angular.isObject -** Determines if a reference is an Object
- **angular.isString -** Determines if a reference is a String
- **angular.isUndefined -** Determines if a reference is undefined
- **angular.lowercase -** Converts the specified string to lowercase
- **angular.uppercase -** Converts the specified string to uppercase

22 $http:

$http is Angular's wrapper around XmlHttpRequest. It provides a set of high level APIs and a low level API to talk to REST services. Each of the API methods return $q promise object.

Following are the APIs exposed by $http:

- **$http.$get(url):** Sends an HTTP GET request to the URL specified
- **$http.post(url, dataToBePosted):** Sends an HTTP POST request to the URL specified
- **$http.put(url, data):** Sends an HTTP PUT request to the URL specified
- **$http.patch(url, data):** Sends an HTTP PATCH request to the URL specified
- **$http.delete(url):** Sends an HTTP DELETE request to the URL specified
- **$http(config):** It is the low level API. Can be used to send any of the above request types and we can also specify other properties to the request. Following are the most frequently used config options:
  - **method:** HTTP method as a string, e.g., 'GET', 'POST', 'PUT', etc.
  - **url:** Request URL
  - **data:** Data to be sent along with request
  - **headers:** Header parameters to be sent along with the request
  - **cache:** caches the response when set to true

Following is a small snippet showing usage of $http:

```
$http.get('/api/data').then(function(result){
  return result.data;
  }, function(error){
    return error;
});
```

INTERMEDIATE - ADVANCED

23 Manage Dependencies

Use a package management tool like *Bower* (bower.io/) to manage and update third-party web dependencies in your project. It is as simple as installing bower using `npm install bower;` then listing all the dependent libraries and versions in a Bower package definition file called bowerconfig.json and lastly run `bower install` or `bower update` in your project to get the latest versions of any web dependencies in your project.

24 Using AngularJS functions

Wherever possible, use AngularJS versions of JavaScript functionality. So instead of setInterval, use the `$interval` service. Similarly instead of setTimeout use the `$timeout` service. It becomes easier to mock them or write unit tests . Also make sure to clean it up when you have no use for it. Use the $destroy event to do so:

```
$scope.$on("$destroy", function (event) {
    $timeout.cancel(timerobj);
});
```

25 Services

If you need to share state across your application, or need a solution for data storage or cache, think of *Services*. Services are singletons and can be used by other components such as directives, controllers, filters and even other services. Services do not have a scope of their own, so it is permissible to add eventlisteners in Services using `$rootScope`.

26 Deferred and Promise

The $q service provides deferred objects/promises.

- **$q.all([array of promises]) -** creates a Deferred object that is resolved when all of the input promises in the specified array are resolved in future
- **$q.defer() -** creates a deferred object with a promise property that can be passed around applications, especially in scenarios where we are integrating with a 3rd-party library

- **$q.reject(reason) -** Creates a promise that is resolved as rejected with the specified reason. The return value ensures that the promise continues to the next error handler instead of a success handler.

- **deferredObject.resolve -** Resolves the derived promise with the value

- **deferredObject.reject -** Rejects the derived promise with the reason and triggers the failure handler in the promise.

27 Manipulating $scope

Do not make changes to the $scope from the View. Instead do it using a Controller. Let's see an example. The following piece of code controls the state of the dialog property directly from the ng-click directive.

```
<div>
   <button ng-click="response = false">Close Dialog
   </button>
</div>
```

Instead we can do this in a Controller and let it control the state of the dialog as shown here:

```
<div>
   <button ng-click="getResponse()">Close Dialog
   </button>
</div>
```

In dialog-controller.js file, use the following code:

```
dialog.controller("diagCtrl", function ($scope) {
    $scope.response = false;

    $scope.getResponse = function () {
        $scope.response = false;
    }
});
```

This reduces the coupling between the view and controller

28 Prototypal Inheritance

Always have a '.' in your ng-models which insures prototypal inheritance. So instead of

```
<input type="text" ng-model="someprop">
```

use

```
<input type="text" ng-model="someobj.someprop">
```

29 Event Aggregator:

$scope includes support for event aggregation. It is possible to publish and subscribe events inside an AngularJS application without need of a third party library. Following methods are used for event aggregation:

- **$broadcast(eventName, eventObject):** Publishes an event to the current scope and to all children scopes of the current scope

- **$emit(eventName, eventObject):** Publishes an event to the current scope and to all parent scopes of the current scope

- **$on(eventName, eventHandler):** Listens to an event and executes logic inside eventHandler when the event occurs**.**

30 $resource

$resource is a higher level service that wraps $http to interact with RESTful APIs. It returns a class object that can perform a default set of actions (get (GET), save (POST), query (GET), remove(DELETE), delete (DELETE)). We can add more actions to the object obtained.

```
//A factory using $resource
myModule.factory('Student', function($resource){
    return $resource('/api/students',  null, {
        change: {method: 'PUT'}
    });
});
```

The above operation returns a $resource object that has all default operations and the change method that performs a PUT operation.

31 $timeout and $interval

$timeout is used to execute a piece of code after certain interval of time. The $timeout function takes three parameters: function to execute after time lapse, time to wait in milliseconds, a flag field indicating whether to perform $scope.$apply after the function execution.

```
$timeout(function () {
    //Logic to execute
}, 1000, true);
```

$interval is used to keep calling a piece of code repeatedly after

certain interval. If count is not passed, it defaults to 0, which causes the call to happen indefinitely.

```
$interval(function () {
  //Logic to execute
}, 1000, 10, true);
```

32 jqLite and jQuery

AngularJS uses a lighter version of jQuery called jqLite to perform DOM manipulations. The element we receive in compile and link functions of directive are jqLite objects. It provides most of the necessary operations of jQuery. Following snippet shows obtaining a jqLite object for all divs on a page using selector:

```
var divJqliteObject = angular.element('div');
```

But, if jQuery library is referred on the page before referring AngularJS, then Angular uses jQuery and all element objects are created as jQuery objects.

If a jQuery plugin library is referred on the page before referring AngularJS, then the element objects get capabilities of the extended features that the plugins bring in.

33 ngCookie:

ngCookie is a module from the AngularJS team that wraps cookies and provides an easier way to deal with cookies in an AngularJS application. It has two services:

i. **$cookieStore:** Provides a key-value pair kind of interface to talk to the cookies in the browser. It has methods to get value of a stored cookie, set value to a cookie and remove a cookie. The data is automatically serialized/de-serialized to/from JSON.

ii. **$cookies:** An object representing the cookies. Can be used directly to get or set values to cookies

34 Unit testing:

AngularJS is built with unit testing in mind. Every component defined in Angular is testable. Dependency injection is the key factor behind it. Take any kind of component in Angular, it can't be written without getting some of the external components injected in. This gives freedom to programmers to pass any object of their choice instead of the actual component object while writing tests. The only thing to be taken care is to create an object with the same shim as the component.

AngularJS code can be unit tested using any JavaScript Unit Testing framework like QUnit, Jasmine, Mocha or any other framework. Jasmine is the most widely used testing framework with Angular. Tests can be run anywhere, in browser or even in console using Karma test runner.

The main difference between application code and unit tests is, application code is backed by the framework and browser, whereas unit tests are totally under our control. So, wherever we get objects automatically injected or created by AngularJS, these objects are not available in unit tests. They have to be created manually.

35 Bootstrapping a unit test:

Just like the case of Angular application, we need to bootstrap a module in unit tests to load the module. As the module has to be loaded fresh before any test runs, we load module while setting up the tests. In Jasmine tests, setup is doe using beforeEach block.

```
beforeEach(function(){
  module('myModule');
});
```

36 Creating $scope in unit tests:

$scope is a special injectable in AngularJS. It is unlike other objects as it is not already created to pass into a component when asked. A $scope can be injected only inside controllers and for every request of $scope, a new $scope object is created that is inherited from $rootScope. Framework takes care of creating the scope when the application is executed. We have to do it manually to get a $scope object in tests. Following snippet demonstrates creation of $scope:

```
var scope;

beforeEach(inject(function ($rootScope) {
    scope = $rootScope.$new();
}));
```

37 Testing controllers:

In an AngularJS application, we generally don't need to create an object of a controller manually. It gets created whenever a view loads or the template containing an ng-controller directive loads. To create it manually, we need to use the *$controller* service. To test the behavior of controller, we need to manually create object of the controller.

```
inject(function($controller){
  var controller = $controller('myController',
  { $scope: scope, service: serviceMock });
});
```

As we see, arguments to the controller are passed using a JavaScript object literal. They would be mapped to right objects according to names of the services. After this, the scope would have all properties and methods that are set in the controller. We can invoke them to test their behavior.

```
it('should return 10', function(){
  var val = scope.getValue();
  expect(val).toEqual(10);
});
```

38 Testing services:

Getting object of a service is easy as it is directly available to the inject() method.

```
var serviceObj;
beforeEach(inject(function (myService) {
  serviceObj = service;
}));
```

Now any public method exposed from the service can be called and the result can be tested using assertions.

```
it('should get some data', function(){
  var data = serviceObj.getCustomers();
  expect(data).not.toBe(null);
  expect(data).not.toBe(undefined);
});
```

39 ng-controller outside ng-view:

Though controllers are used with views in general, it doesn't mean that they can't be used outside a view. A controller can be made responsible to load menu items, show toast messages, update user when a background task is completed or any such thing that doesn't depend on the view loaded inside ng-view.

```
<div ng-app="myModule">
    <div ng-controller="menuController">
        <!-- Mark-up displaying Menu -->
    </div>
    <div ng-view></div>
</div>
```

40

To avoid controllers from getting too complicated, you can split the behavior by creating Nested Controllers. This lets you define common functionality in a parent controller and use it one or more child controllers. The child controller inherits all properties of the outside scope and in case of equality, overrides the properties.

```
<body ng-controller="mainCtrller">
  <div ng-controller="firstChildCtrller">
  </div>
</body>
```

41 Mocking services:

Mocking is one of the most crucial things in unit testing. It helps in keeping the system under test isolated from any dependency that it has. It is very common to have a component to depend on a service to get a piece of work done. This work has to be suppressed in unit tests and replaced with a mock or stub. Following snippet mocks a service:

Say, we have following service:

```
app.service('customersSvc', function(){
  this.getCustomers = function(){
    //get customers and return
  };
  this.getCustomer = function(id){
    //get the customer and return
  };
  this.addCustomer = function(customer){
    //add the customer
  };
});
```

To mock this service, we need to create a simple object with three mocks with the names same as the ones in the service

and ask Angular's injector to return the object whenever the service is requested.

```
var mockCustomersSvc;

beforeEach(function(){
  mockCustomerService = {
    getCustomers: jasmine.createSpy('getCustomers'),
    getCustomer: jasmine.createSpy('getCustomer'),
    addCustomers: jasmine.createSpy('addCustomer')
  };

  module(function($provide){
    $provide.value('customersSvc', mockCustomersSvc);
  });
});
```

42 ngMock

The ngMock module provides useful tools for unit testing AngularJS components such as controllers, filters, directives and services.

- The *module* function of ngMock loads the module you want to test and it's inject method resolves the dependencies on the service to be tested

- We can mock the backend and test components depending on the *$http service* using the *$httpBackend* service in ngMocks

- We can mock timeouts and intervals using *$interval* and *$timeout* in ngMocks

- The *$log* service can be used for test logging

- The *$filter* service allows us to test filters

- Directives are complex to test. Use the $compile service and *jqLite* to test directives

43 ng-class:

ng-class can be used in multiple ways to dynamically apply one or more CSS classes to an HTML element. One of the very good features is, it supports some simple logical operators too. Following list shows different ways of using ng-class:

i. `<div ng-class="dynamicClass">some text</div>`

Assigns value of dynamicClass from scope to the CSS class. It is two-way bound, i.e. if value of dynamicClass changes, the style applied on the div also changes.

ii. `<div class="[class1, class2, class3]">some text</div>`

All classes mentioned in the array are applied

iii. `<div class="{''my-class-1'':value1, ''my-class-2'':value2}">some text</div>`

my-class-1 is applied when value1 is assigned with a truthy value (other than false, empty string, undefined or null)

iv. `<div ng-class="value ? 'my-class-1':'my-class-2'">some text</div>`

Value of class applied is based on result of the ternary operator.

v. `<div ng-class="{true: 'firstclass'}[applyfirstclass] || {true:'secondclass'}[applysecondclass]"></div>`

Here, applyFirstClass and applySecondClass are data bound variables. The expression applies firstClass if applyFirstClass is true. It applies secondClass only if applySecondClass is true and applyFirstClass is false.

44 Resolve blocks in routing:

Resolve blocks are used to load data before a route is resolved. They can be used to validate authenticity of the user, load initial data to be rendered on the view or to establish a real-time connection (e.g. Web socket connection) as it would be in use by the view. View is rendered only if all the resolve blocks of the view are resolved. Otherwise, the route is cancelled and the user is navigated to the previous view.

```
$routeProvider.when('/details', {
  templateUrl: 'detailsView.html',
  controller: 'detailsController',
  resolve: {
    loadData: function (dataSvc, $q) {
      var deferred = $q.defer;
      dataSvc.getDetails(10).then(
      function (data) { deferred.reslve(data);},
       function () { deferred.reject();});
            return deferred.promise;
    }
  }});
```

In the above snippet, the route won't be resolved if the promise is rejected. Resolve block can be injected into the controller and data resolved from the resolve block can be accessed using the injected object.

45 $compile

Used to compile templates after the compilation phase. $compile is generally used in link function of directives or services. But, it should be used with caution as manual compilation is an expensive operation.

```
myModule.directive('sampleDirective', function(){
  return {
    link: function(scope, elem, attrs){
      var compiled = $compile('<div>{{person.name}}
      </div>')(scope);
                    elem.html(compiled);
    }
  };
});
```

46 $parse

$parse is used to transform plain text into expression. The expression can be evaluated against any object context to obtain the value corresponding to the object. Very common usage of parse is inside directives to parse the text received from an attribute and evaluate it against scope. The expression also can be used to add a watcher.

```
myModule.directive('sampleDirective', function($parse){
  return function(scope, elem, attrs){
    var expression = $parse(attrs.tempValue);

    var value = expression(scope);
    scope.$watch(expression, function(newVal, oldVal){
      //Logic to be performed
    });
  };
});
```

47 Route change events:

When a user navigates from one page to another, AngularJS broadcasts events at different phases. One can listen to these events and take appropriate action like verifying login status, requesting for data needed for the page or even to count the number of hits on a view. Following are the events raised:

- $routeChangeStart
- $routeChangeSuccess
- $routeChangeError
- $routeUpdate

48 Decorating

It is possible to modify the behavior or extend the functionality of any object in AngularJS through decoration. Decoration is applied in AngularJS using $provide provider. It has to be done in config block. Following example adds a method to the value:

```
angular.module('myModule',[])
  .config(function($provide) {
    $provide.decorator('someValue', function($delegate)
    {
      $delegate.secondFn = function(){
        console.log("Second Function");
      };

      return $delegate;
    });
  })
  .value('someValue',{
    firstFn: function(){console.log("First Function");}
  });
```

Note: *Constants cannot be decorated*

49 Exception handling

All unhandled exceptions in an AngularJS application are passed to a service $exceptionHandler, which logs the error message in the browser's console. In large business applications, you may want to log the error details on the server by calling an API. This can be done by decorating the $exceptionHandler service.

```
myApp.config(function ($provide) {
  $provide.decorator('$exceptionHandler', ['$log',
  '$http', '$delegate',
    function ($log, $http, $delegate) {
      return function (exception, cause) {
        $log.debug('Modified exception handler');
        $http.post('/api/clientExceptionLogger',
        exception);
        $delegate(exception, cause);
      };
    }
  ]);
});
```

50 HTTP Interceptors

Any HTTP request sent through $http service can be intercepted to perform certain operation at a given state. The state may be one of the following: before sending request, on request error, after receiving response and on response error. Interceptors are generally used to check authenticity of the request before sending to the server or to displaying some kind of wait indicator to the user when the user has to wait for the data to arrive. The intercepting methods may return either plain data or a promise.

```
myModule.config(function ($provide) {
  $provide.factory('myHttpInterceptor', function () {
    return {
      request: function (req) {
        //logic before sending request
      },
      response: function (res) {
        //logic after receiving response
      },
      requestError: function () {
        //logic on request error
      },
      responseError: function () {
        //logic on response error
      }
    };
  });
});
```

51 HTTP Transforms

Transforms allow us to tweak the data before sending to an HTTP request or after receiving response at the end of a request. It can be applied either globally using a config block or on a specific request using the $http config object.

Setting transform in config block:

```
myModule.config(function ($httpProvider) {
  $httpProvider.defaults.transformRequest.push(function
  (data) { //Operate on data });
});
```

In the individual request:

```
$http({
  url: '/api/values',
  method: 'GET',
  transformRequest: function (data) {//Operate on data}
});
```

Some useful AngularJS Tools & Libraries

- Visual Studio Express (free), Sublime (paid), NetBeans (free) or WebStorm (paid) for AngularJS Development
- Batarang Chrome Development Plugin (bit.ly/dncm15-batarang) for debugging and profiling AngularJS applications
- AngularUI (angular-ui.github.io/) and ngModules (ngmodules.org) are a good place to look out for commonly used modules
- Angular Seed (bit.ly/dncm15-angularseed), ngBoilerplate (github.com/ngbp/ngbp) and Yeoman (yeoman.io/) can be used for workflow and project structure management
- The Iconic Framework (http://ionicframework.com) is an Angular wrapper around PhoneGap; can be used to build hybrid mobile apps with Angular

Some useful companion libraries in AngularJS

- AngularUI-Bootstrap (github.com/angular-ui/bootstrap) provides native AngularJS directives for Bootstrap (No need of jQuery)
- AngularUI-Utils (github.com/angular-ui/ui-utils) contains a bunch of essential utilities (included with ng-boilerplate)
- AngularUI (angular-ui.github.io/) ports off many jQueryUI components to Angular
- Angular translate (angular-translate.github.io) makes it easier to apply internationalization to Angular projects
- Restangular (github.com/mgonto/restangular) A clean, promise-based, feature rich 3rd-party library that provides some useful abstractions, especially for complex operations on the client-side.
- AngularFire (firebase.com/docs/web/libraries/angular/): An abstraction to interact with Firebase realtime database
- Breeze.js (breezejs.com): A library for rich data operations in AngularJS apps. Also contains directives for data validations ■

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

PERFORMANCE OPTIMIZATION

IN ASP.NET WEB SITES

Performance is an important aspect of the modern day web application development. Not only does it make a site seamless to use but also increases the scalability of the website and makes it future proof. In this article, we will look at various aspects of improving the performance of ASP.NET web applications. We will only concentrate on the browser/web server side performance as opposed to server/app server/database server performance optimizations.

Before we get into the details, the first question is, do we really need to optimize web sites?

Here are some examples on why you would need optimized web sites.

- Amazon.com had performed a test on their web site, and when they slowed the site by 100ms, the sales drop was 1%. As you would imagine for a company like Amazon, 1% is a huge loss.

- Google slowed their Search Engine by 500ms, and the traffic dropped by 20%.

As you can see, performance is a very important aspect of modern web sites. It becomes more important, as nowadays most sites require optimized sites for mobile, tablet devices and other portable devices, that often run on low throughput wireless networks.

Here are some tips that you can consider while making a better performing website.

USING THE RIGHT ASP.NET FRAMEWORK

Check your .NET framework. If you can upgrade your site to use .NET 4.5, then it has some great performance optimizations. .NET 4.5 has a new Garbage Collector which can handle large heap sizes (i.e tens of gigabytes). Some other improvements are Multi-core JIT compilation improvements, and ASP.NET App Suspension. These optimizations do not require code changes. A great article on an overview of performance improvements in .NET 4.5 is at the following url:

http://msdn.microsoft.com/en-us/magazine/hh882452.aspx

FILE COMPRESSION

There are often requests bloated to the web server with lot of static content. These content can be compressed thereby reducing the bandwidth on requests.

The following setting is only available in II7 and later.

```
<configuration>
  <system.webServer>
    <urlCompression doStaticCompression="true"
    doDynamicCompression="true" />
  </system.webServer>
</configuration>
```

The above configuration setting has direct association with IIS and nothing to with ASP.NET. The *urlCompression* name sounds strange but it is not really the compressing of URLs. It means compressing or *gzipping* the content that is sent to the browser. By setting to true/enabling, you can gzip content sent to the browser while saving lots of bandwidth. Also notice that the above settings does not only include static content such as CSS/JS, but also dynamic content such as .aspx pages or razor views.

If your webserver is running in Windows Server 2008 R2 (IIS7.5), these settings are enabled by default. For other server environments, you would want to tweak the configuration as I just showed, so that you can take the advantage of compression.

REDUCING THE NUMBER OF REQUESTS TO THE SERVER

It is very common that lot of websites use individual CSS files and JS files as static content. Usually each file is served per request. For a small site this seems minimal, but a number of large static files, when requested via the web server, utilizes lot of bandwidth over the network.

ASP.NET provides a great way to bundle and minify these static content files. This way the number of requests to the server can be reduced.

There are many ways to bundle and minify files. For example, MSbuild, third party tools, Bundle Configs etc. But the end result is the same. One of the easiest ways is to use the new Visual Studio Web Essentials Pack. You can download this extention from here http://vswebessentials.com/

Once installed, you can create minified and bundled CSS and Script files using the Web Essential Pack as shown here:

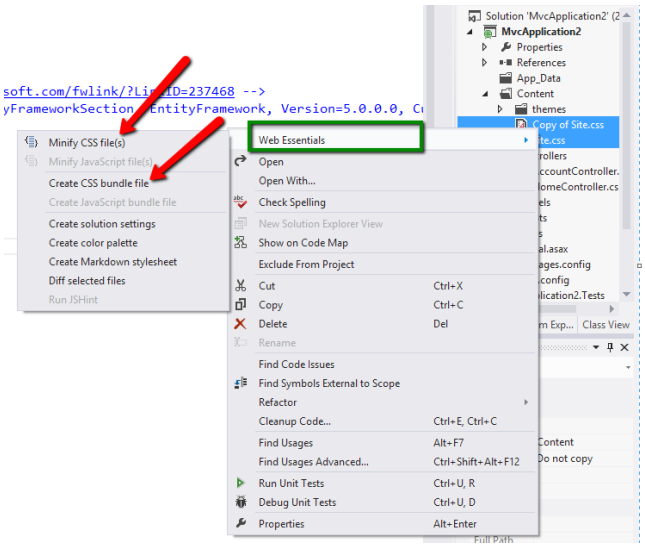


Figure 1: Web Essentials Minification and Bundling

The above menu items would create bundled/minified files, and add them to your project. Now instead of referencing multiple CSS and JS files, you can simply reference the minified and bundled versions. This would also mean that there will be lesser requests made to the server, which will result in less bandwidth and faster responses.

You may wonder that even if we have bundled all files together, the 'total' size of these files would remain the same, as compared to when we were to serve them individually. However this is not necessarily true. For example,

a. The minified process would reduce the size of the files by removing comments, shortening variable, and removing spaces etc.

b. Bundling them together would also remove the additional Http headers that is required for each individual request.

You can also find some good information about bundling and minification in the following link http://www.asp.net/mvc/tutorials/mvc-4/bundling-and-minification

GET CONTROL OF YOUR IMAGE REQUESTS

It is possible to reduce the number of requests for images.

There are couple of ways to do this.

a. You may create an Image Sprite. With image sprite, you can combine multiple different images into a single large image. Then use CSS to reposition those images within the site.
The following article by ASP.NET MVP Abhimanyu shows how you can create Image Sprites using web essentials http://www.itorian.com/2014/02/creating-image-sprite-in-visual-studio.html

b. Base64 Data URIs
With this option, you would never make any requests to the server to obtain any images. In Visual Studio 2013, you can take your smaller images and directly embed them into you CSS/ Stylesheet.

So

```
.main-content {
    background: url("../Images/accent.png") no-repeat;
    padding-left: 10px;
    padding-top:
}
```

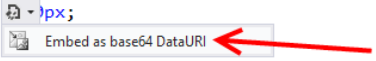


Figure 2: Embed as base64 DataURI

will become..

```
.main-content {
  background: url('.....+ ')
  padding-left: 10px;
  padding-top: 30px;
}
```

SETTING UP EXPIRY HEADERS

By default, static content served from the web server does not have expiry dates. We can instruct the browser on how to cache the content, and for how long.

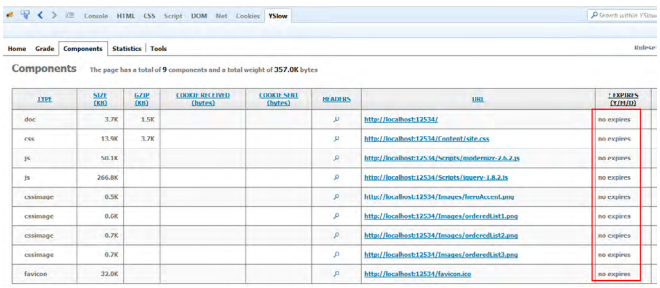


Figure 3: Expiry Headers

If you set the expiration to a future date, the browser will not make a request to the server but instead the static content will be served from the browser's internal cache.
In the web.config, there is a section where you can control these settings.

```
<configuration>
  <system.webServer>
    <staticContent>
      <clientCache cacheControlMode="DisableCache" />
    </staticContent>
  </system.webServer>
</configuration>
```

We can change these settings to cache all the static content requested from the server, for instance cache it maximum for a year. Note the expiration is a sliding expiration, which means the content will be served from the cache from today up to a year.

```
<configuration>
  <system.webServer>
    <staticContent>
      <clientCache cacheControlMode="UserMaxAge"
      httpExpires="365.00:00:00" />
    </staticContent>
  </system.webServer>
</configuration>
```

Now the web server will automatically add this header to the static files.

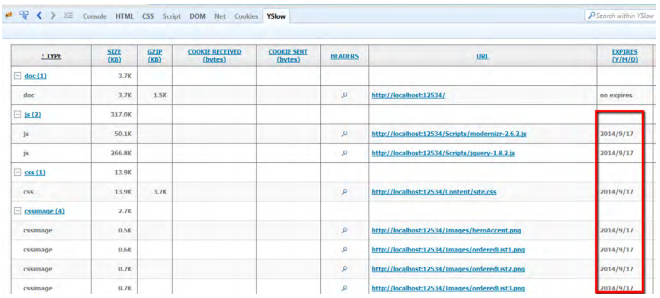


Figure 4: Sliding Expiration

*(**Note:** The above Type 'doc(1)' is the Html dynamic file hence the cache settings are not applied.)*

This is all good, but what would happen if you make a change to your CSS/JS file. With the above settings in place, the browser

will not serve those changes for the entire year. One way to tackle this issue is to force the browser to refresh the cache. You may also change the URL (i.e add query string, fingerprint/timestamp) to treat the page as a new URL, so the cache gets refreshed.

SCRIPT RENDERING ORDER

If possible, you can move the script tags <script> to the very bottom of the page. The reason this is important is because during the rendering, when the browser comes across a <script> tag, it stops to process the script and then moves ahead. If you put the script tags at the bottom of the page, the page/HTML will render faster and the scripts can execute after the DOM elements have loaded.

Sometimes moving the script to the bottom of the page is not possible as some DOM elements or CSS may depend on these scripts, so they can be rendered. In such cases, you could move those scripts further up the page. However, as a rule of thumb, try to keep the scripts as lower towards the bottom as possible. Positioning the <script> tag towards the bottom of the page is not the only option to defer the load of script files. There are other ways too, for example, you can use the *defer* attribute.

```
<script src=”some.js” defer></script>
```

By using the defer attribute, you can specify the script not to run until the page has been fully loaded. Check out some limitations of the defer attribute https://developers.google.com/speed/pagespeed/service/DeferJavaScript

Another way is to configure your scripts to run asynchronously.

```
<script src=”some.js” async></script>
```

Using the above ***async*** tag, the scripts will be run asynchronously, as soon as it is available.

Defer and async do almost the same thing, i.e. allow the script to be downloaded in the background without blocking. The real difference is that 'conceptually' *defer* guarantees that scripts execute in the order they were declared on the page, which means although some scripts may be downloaded sooner than the others, they have to wait for scripts that were declared before them.

By the way, there's nothing stoping you from doing this

```
<script defer async src=”...”></script>
```

in which case, async overrides defer.

OPTIMIZING IMAGES

Images are static content and they do take some bandwidth when requested via a web server. One way to solve this is to reduce the size of the images, in other words optimize the images.

Image "Optimization" does not mean that it reduces the quality of the image. But it will re-arrange the pixels and palettes to make the overall size smaller.

Web Definition of Image Optimization is "This term is used to describe the process of image slicing and resolution reduction. This is done to make file sizes smaller so images will load faster."

So how you optimize images?
There are many third-party tools that can optimize images. The following VS extension would do the trick for you - http://visualstudiogallery.msdn.microsoft.com/a56eddd3-d79b-48ac-8c8f-2db06ade77c3.

Alternatively if you are using png's in your site, tinypny.org is a good place to reduce the file size of those images.

Size of favicon: This is often ignored but you may have not noticed that sometimes the size of favicon is considerably large. A favicon for a website normally has to be a cacheable *.ico file and can be 32x32 or 16x16. You can also read http://stackoverflow.com/questions/1344122/favicon-png-vs-favicon-ico-why-should-i-use-png-instead-of-ico for a good discussion on which format to use and why.

CACHING HTML

If you have pages that never get updated, you can cache those pages for a certain period. For example, if you use Web Forms or ASP.NET MVC, you could use ASP.NET Output Caching.

http://msdn.microsoft.com/en-us/library/xsbfdd8c(v=vs.90).aspx

With ASP.NET Web Pages you can modify the response headers for caching. Here are some caching techniques to follow for web pages.

http://msdn.microsoft.com/en-us/library/vstudio/06bh14hk(v=vs.100).aspx

http://www.asp.net/web-pages/tutorials/performance-and-traffic/15-caching-to-improve-the-performance-of-your-website

It is important to note that this would only work if you have pages that *do not* require frequent updates.

TOOLING SUPPORT FOR OPTIMIZING WEB SITES

I believe that this is the most important aspect of this article, as the right tool goes a long way in producing optimized web sites. There are a number of tools, but there two of them that always stand out.

1. **Yahoo's YSlow** (https://developer.yahoo.com/yslow/)

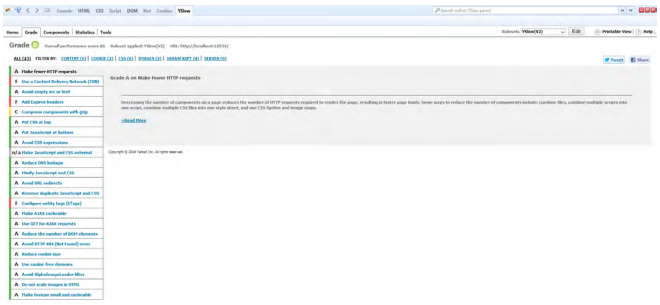


Figure 5: Yahoo's YSlow in action

It is a Firefox Add-On and one of the most popular tool among web developers.

YSlow has some great features including a grading system, and instructions to make your site optimized. Most of the tips I described in this article are also provided as tips in this tool, so once your site is optimized, your grading should go up.

2. **Google's PageSpeed**

This is another great Chrome browser extension. Once installed, it is also available as a part of the chrome developer tool bar.

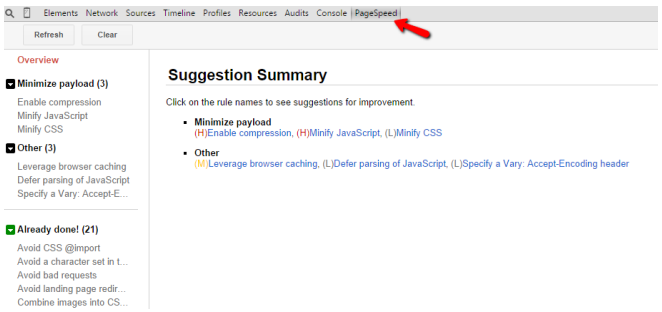


Figure 6: Google's PageSpeed

There is no guarantee that you could make all the optimizations that these tools suggest. At the end, it is all based on the website, and requirements you need to support. But combining both of these tools to measure your site's performance and applying some of the tips I mentioned earlier, is a great start.

CONCLUSION

In this article, we looked at some of the approaches that you can take to make optimized and better performing web sites. This included, reducing number of requests to the server, changes to the .NET framework, compressing techniques and finally some tools that allow you to make better decisions on optimizing web site performances.

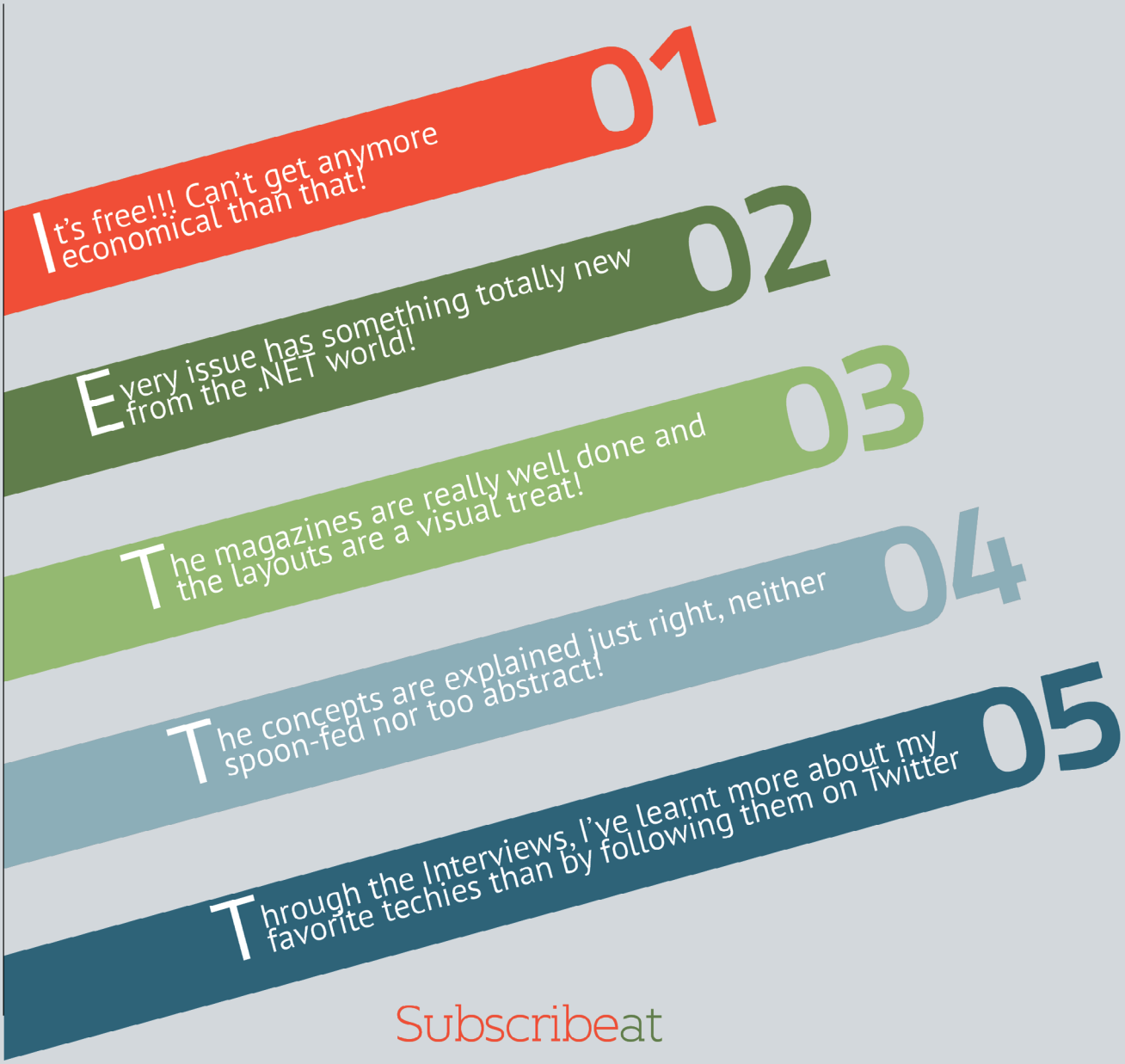
It is also important to consider that whilst these techniques optimize web site/ web server performances, the overall performance can be based on a number of other factors. This includes performance of Application Server, Database Server (in a multi-tiered setup) and so on. However to start with, there are a lot of performance improvements you can gain by optimizing sites as described in this article ■

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

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

DNC MAGAZINE

(IF YOU HAVEN'T ALREADY)



Subscribeat
www.dotnetcurry.com/magazine

HARVESTING

Anyone that has a vegetable garden knows the importance of harvesting the veggies at exactly the right time. If you pick a vegetable too soon, it won't have good flavor. Pick it too late and it will be soft, mushy, and rotten. Just like the gardener, we need to harvest our software applications when they are at the height of ripeness. Too soon and the app will be full of bugs. Too late and we'll have lots of *feature creep* or the business needs will have changed and the application will be full of rotting code.
How do you determine the right time to harvest or release your application? Applying Agile, you release often. In fact, every two weeks is very common. This schedule helps you know that the application is not overly ripe. But that doesn't help you know that application is actually properly ripened and ready for release. What does help you know if the code is ready for release, is by applying **Continuous Delivery techniques**.

Continuous Delivery uses what is called the "deployment pipeline". Code goes in one side and fully tested applications come out the other. The idea of the deployment pipeline is that the further to the left you find a bug, the less costly it is to fix. This is commonly called "fail fast".

Figure 1. The Deployment Pipeline

The other idea of the Deployment Pipeline is that code progresses in an automated fashion from left to right. This means that you automate deployments and automate testing, as much as possible. In fact, you can automate almost all the testing including unit, regression, integration, capacity, acceptance, functional, system, stress, performance, and deployment. Manual testing includes things such as exploratory, usability, showcases, look and feel, and worst-case testing.

You can see that Harvesting is a large topic. I only have space to discuss one aspect, so let's talk about the first step, Continuous Integration. It's here that your code is integrated with code from your co-workers and then unit tested to ensure the minimal functionality is still, well, functional. Ideally, the integration happens with each check-in to version control and takes five minutes or less to complete.

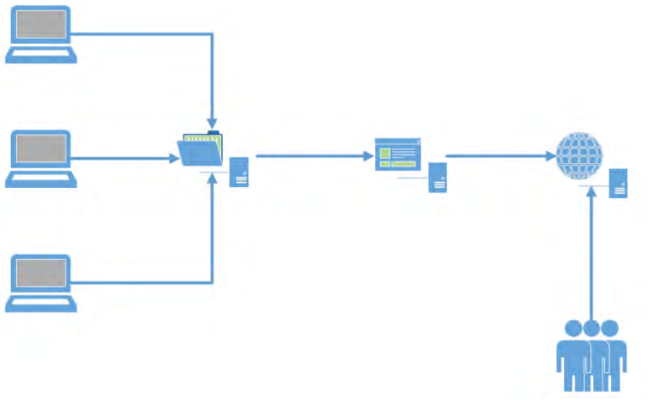


Figure 2. The Continuous Integration process

Using Continuous Integration (CI), each developer builds and tests their code locally. When the functionality is complete and their tests pass, updated code that has been written by other team members is brought locally and merged. This is the integration step. Tests are then run again locally to ensure code still passes. This is important. Once you get the unit tests to pass, check in the code. Ideally, this happens several times a day. Hence, "continuous integration". Once the code passes this step, you check in your code.

On the other side of the version control system is a server that watches for code check-ins. When one occurs, it grabs the code and compiles it. If the compilation is successful, unit tests are run against the code. This unit testing step is critical as it identifies areas where code from one developer has side effects on code from another developer. The code doesn't integrate. If you don't have unit tests, you don't have Continuous Integration. You only have an automated build.

Some systems, such as TFS, use a "gated check-in". When you check in the code, it's put in isolation and unit tests are again run. Only if those tests pass, is the code allowed into the version control repository. In this column, I will show you how to setup Team City for your CI server. Because Team City does not have its own source repository, it cannot force code to pass tests before check-in. However, Team City does support a "personal build" that you can run before check in. Note I will not show the personal build, but it is fully documented on the Team City web site.

The outputs of the CI build and test process are typically assemblies and other files. These are called "artifacts" in CI parlance. Hopefully your tests pass so you can get these artifacts.

The final step is reporting back to the team on the outcome of the integration. This is typically done through a web site. The CI server software is a program designed to make this process easier. There are many CI servers that you can choose from including TFS, Go, Jenkins, TeamCity, and others. I will use Team City for my examples because I believe Team City is best of breed, even surpassing TFS. Why? TFS is very process heavy and you're pretty much tied to how Microsoft thinks things should be done. Team City allows you to pick best of breed for version control, bug tracking, project management, deployment, etc. Depending on which other tools you choose, Team City or the other tool may have built-in integration points with them.

Also, if your team uses languages and tools other than .NET, Team City may work out to be a much less expensive option. The cost is free up to 20 configurations (I'll explain that in a bit), then its $2000 US. Really a small price to pay for a great tool. You may also be thinking that Jenkins, which is always free, is cheaper. Yes, initially it is, but you have to deal with lots more XML files and do more manual configuration, making it more costly in the end.

MSBUILD

Before getting into Team City, we need to talk about MSBuild. Every time you build in Visual Studio, an MSBuild file is used. In fact, .csproj and .vbproj files are really MSBuild files. We'll use a special MSBuild file for Team City.

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="ReleaseBuild"
xmlns="http://schemas.microsoft.com/developer/
msbuild/2003" ToolsVersion="4.0">

  <!-- Solution/project changes here -->
  <PropertyGroup>
    <SolutionName>TeamCity</SolutionName>
    <StartupProjectPath>TeamCity</StartupProjectPath>
    <StartupProjectName>TeamCity</StartupProjectName>
  </PropertyGroup>

  <Target Name="ReleaseBuild">
    <Message Text="Build number is $(BUILD_NUMBER)" />
    <MSBuild Projects="$(SolutionName)
    .sln" Targets="Clean;Build"
    Properties="Configuration=Release"/>
  </Target>
</Project>
```

A bit of explanation is needed. Targets are the sections that do the work. Think of them as procedures. There is one target, ReleaseBuild, in this project. A PropertyGroup is basically a property bag. You can see our solution name is TeamCity. In my example, you will save this file as Build.proj in the same folder as the solution.

You can open a Visual Studio command prompt, navigate to the solution folder and type MSBuild Build.proj and see the output in the console window. Yes, MSBuild is a command line tool. It originally shipped as part of the .Net Framework but as of

Visual Studio 2013 it has been broken out into its own package called Microsoft Build Tools. It gets installed with Visual Studio, but will need to be installed separately on the build server. *Never* install Visual Studio on your build server.

SETTING UP TEAM CITY

I will use the term "build server" when referring to the actual server that runs the CI software, and "CI Server" as the generic term for the CI server software. You can download Team City from the Jet Brains web site. You can connect Team City to use Active Directory or use its own internal security. You also need a database for Team City to store configuration information. It ships with a small database server, but it is recommended you use it only for evaluation purposes. I typically use SQL Server Express for production build servers.

You will also need to install the version of the .NET Framework you are using, the Microsoft Build Tools, and NuGet. These are all free so you don't need to worry about licensing costs.

Team City uses a hierarchy of objects to manage your build. At the top is a project. You can think of this as a Visual Studio solution. You can nest one Team City project inside another if you wish, but I haven't had a need to do this.

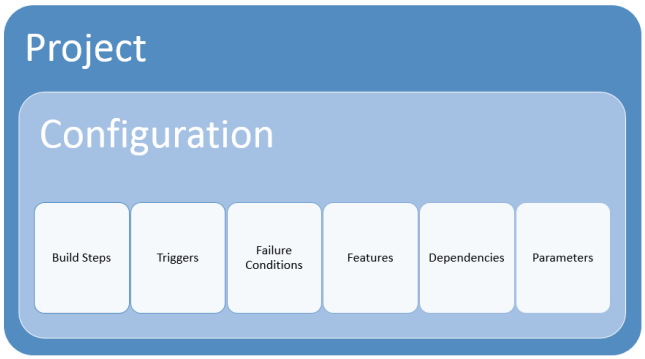


Figure 3. The Team City object hierarchy

Next is the Configuration. This is where the work is done. For example, everything needed to do an Integration Build is a Configuration. If you do a nightly build to run additional tests, you have another Configuration. If you have Team City deploy to your QA server, that's another Configuration. Remember, the free version of Team City supports up to 20 Configurations.

Inside a Configuration, you have several types of objects such as Build Steps (we'll see those in a moment), Triggers that determine when the build runs, Dependencies (one Configuration can depend on another), etc. We'll look at some of these objects as we setup an Integration build.

CREATING THE TEAM CITY PROJECT

Once you have Team City and the other tools installed, navigate to the Team City website on your server. In the upper right-hand corner, click "Administration" then "Create Project". I usually give the Team City project the name of the application I'm working on and then accept the default value for the Project ID. In my example, I named the Team City project DNC even though my Visual Studio solution is named TeamCity.sln.

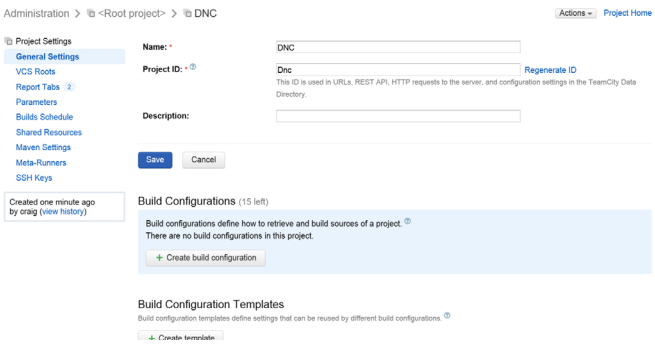


Figure 4. Team City project settings

Looking at the left-hand side of the TeamCity configuration, you'll now see the Project Settings menu. This menu is context sensitive and will change based on where we are in our build configuration.

CONNECTING TO VERSION CONTROL

The first step is setup the connection to version control. Click on VCS Roots then Create VCS root. I've found it's generally easier to select the type of VCS. You'll need to supply the URL to the VCS source, a username, and password. Depending on your version control system, you may need to provide additional configuration data.

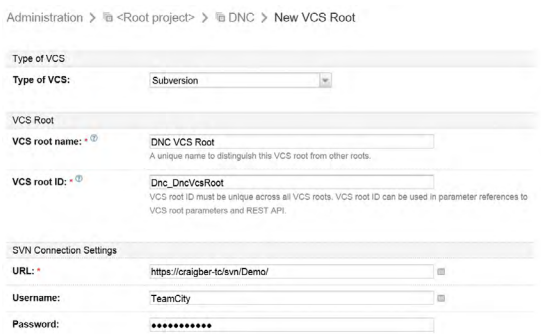


Figure 5.: Configuring the VCS Root

Once you have finished configuring the VCS Root, click Create. You will see a screen listing all the VCS Roots for this project. The next thing is to configure the build.

CONFIGURING THE BUILD

Click General Settings on the Project Settings menu, then click Create build configuration. You will need to enter the name of the build. As this is the integration build, I selected "100 – Integration Build". By numbering the different builds, they will be displayed in the same order that they run. So, if the next build is to deploy to QA, you would name that build "200 – Deploy to QA". Once you've named this build, click Create.

You need to tell this build to use the VCS Root you created earlier. Select it from the dropdown and click Attach. Team City now examines the files in version control and suggests different methods to do the build. I prefer to do this manually. Click the link that says "configure build steps manually" then on the next page, select NuGet Installer as the build runner type.

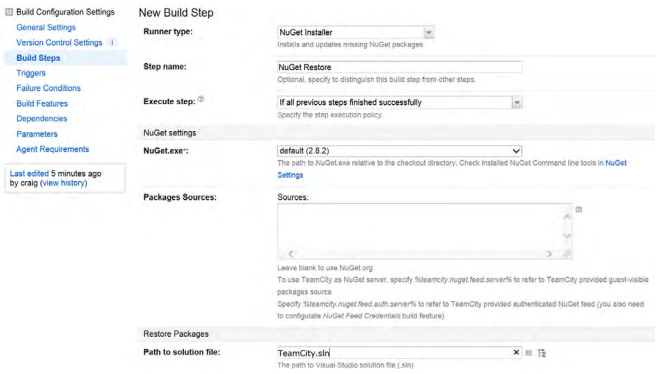


Figure 6: Restoring NuGet files

It's necessary to restore NuGet packages because they shouldn't be checked into version control. It will be impossible to build without these files. In Figure 6, you can see that I named this step NuGet Restore, I selected the version of NuGet to use and entered the path to TeamCity.sln, the Visual Studio solution. Note also that the left context menu has changed to Build Configuration Settings. Save this step then Add a build step. This next step will be the actual build, so select MSBuild as the build runner type.

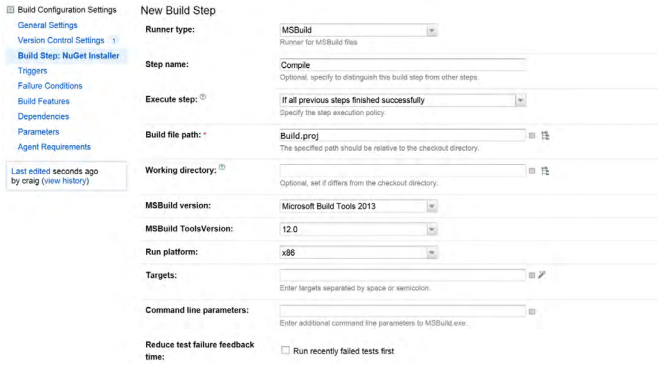


Figure 7: Settings for the Compile build step.

Name the step, point to Build.proj for doing the build, and select the proper MSBuild version. Then click Save. We have one more build step to add. This one will run unit tests. Select NUnit for the build runner type.

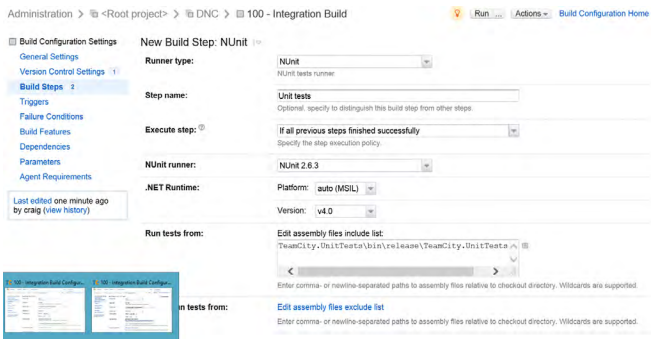


Figure 8: Configuring unit tests

As part of the unit test configuration, you need to enter the path to the unit test .dll file. This path is relative to the root directory of the Visual Studio solution file. Click Save. Team City has pre-build runners for NUnit and MSTest, but you can hook up any unit testing framework that has its own console runner.

SETTING THE VERSION NUMBER

Before we go too much further, let's step back just a bit and add some information about the build itself. Click General Settings. One of the features of Team City is the ability to number the build. Think about this for a moment. If you have five developers, each will have a different build number on their local machine. How will Team City know which version number is correct? Well, because the most recent check-in wins, it will be the more recent version of AssemblyInfo.cs. Let's setup TeamCity to handle the build number. This is done in two steps.

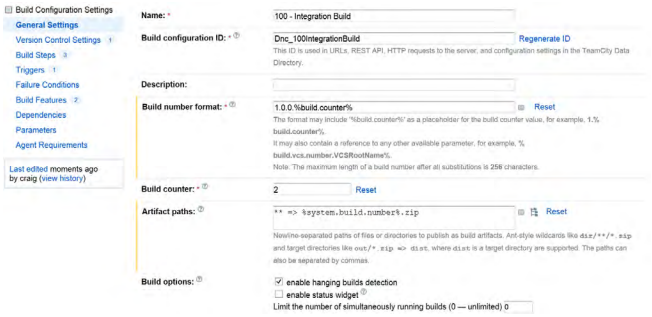


Figure 9: Setting the build number and artifacts

The first step is the build number format. I've used 1.0.0.%build.counter%. Team City will increment the build counter with each build. So, the first build will be 1.0.0.1, the second 1.0.0.2 and so on. When you're ready to work on version 2 of your application, just set the build number to 2.0.0.%build.number% and Reset the build counter.

Before moving to step two, we need to tell Team City to keep the generated artifacts. In the Artifacts paths box, enter ** => %system.build.number%.zip. This tells Team City to take all the files that it checked out and those it created in the build and put them in a zip file with a file name, the same as the build number.

Now click Save.

The second step is to tell TeamCity to update AssemblyInfo.cs with the new build number. On the Build Configuration Settings menu, click Build Features then Add build feature. And select Assembly Info Patcher. Team City will automatically select the proper build number format. Click Save.

TAGGING VERSION CONTROL

There's one more thing we need to configure on this build. It would be nice if we had Team City tag version control with the build number. This way, we always know what version of source control files went into a specific build number. Again, click Build Features and Add build feature. This time, pick VCS Labeling and then select the VCS Root for this project.

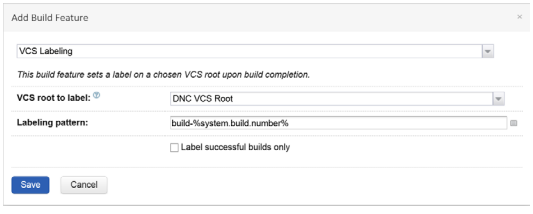


Figure 10: Tagging version control with the build number

Once you configured this feature, click Save.

SCHEDULING THE BUILD

There's one last thing we need to setup. We need to schedule this build to run whenever Team City detects a change in version control. To do this, click Triggers then Add new trigger and select VCS Trigger from the drop down. Note you can also manually run the build or schedule the build to run on a specific day or time.

If you have a busy build server, Team City will queue up the builds and run them in order or you can purchase additional build agents and install them on another server (The Team City Enterprise license gives you three build runner licenses.) Using this scenario, one server hosts Team City and handles farming out the builds to other servers.

Finally, the only thing left is to run the build. There are a couple of ways to do this. You can modify a file in the Visual Studio project and check it in or you can manually trigger the build. Return to the Team City home page, expand the DNC project and click the Run button for the build configuration we just created.

Team City will go out to version control, get the source files, copy them locally, then run each build step in order. If all went well, you'll see the build result will be green. If not, it will be red, and you'll have to drill into the build results to determine what went wrong.

Developers can get results in three different ways. First is a system tray application that simply shows a red/green condition for all builds you are monitoring. The second is a Visual Studio plug-in. Third is by going to the Team City site of your build server and looking at the build status. If a build fails for any reason, this is where you go to get the results. You can drill-down to the details.

Figure 11: Results of the build on the Team City home page

One final note. You can also hook up additional tools like StyleCop or FxCop, but I don't recommend these for an Integration build. Why? FxCop is a bit outdated and doesn't support .Net 4.x very well. Additionally, Team City includes its

own inspections module that does the same basic job (this is the same inspections that you can run from Resharper). The second reason for not running FxCop is the same reason I wouldn't run StyleCop, SQLCop, etc as part of the integration build…they take time. Ideally, an integration build should be run in five minutes or less and running these additional pieces adds time to the build.

Instead, create an Inspections Build and run it at night. This is when you run inspections such as StyleCopy, TeamCity Inspections, SQLCop, or any other review tool you wish. In other words, inspections are not part of Continuous Integration and out of scope for this instance of Software Gardening. I'll come back and address them in the future.

So, how important is Continuous Integration? Well, the Agile Manifesto says you are to deliver working software. If you are trying to deliver every two weeks, there is no way you can deliver working software without using CI. It would be near impossible to determine everything is working correctly without it.

Hopefully you understand what CI can do for you and how easy it is to setup in Team City. After all, it is only by using CI as the first step in your deployment pipeline that you can ensure your software stays lush, green, and vibrant ■

ABOUT SOFTWARE GARDENING

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

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.

CUSTOMIZING TEST PLAN AND TEST SUITE WORK ITEMS

using Visual Studio 2013 Update 3 and Microsoft Test Manager 2013

In the previous versions of Team Foundation Server and Visual Studio; Test Plan and Test Suites were entities independent of work item tracking service. Since they were not work items, it was not possible to add a state or any field to it. This lead to stringent workflows. Test Case could be customized but the Test Suite or Test Plan were not customizable. It was a long awaited requirement that Test Plan and Test Suite be provided as a work item. This was needed in order to customize Test Plan and Test Suite artefact's. Visual Studio 2013 Update 3 now provides Test Plan and Test Suite as work items. Now we can do any customization to them like adding fields, adding state transitions and providing workflow. This can be used to create custom reports by marking the newly added fields as Reportable. We can even have versioning and history for them.

In this article, we will discuss benefits of Test Plan and Test Suites as work items, which can now be customized to suit the specific needs of an organization. We will find out how to change the workflow of Test Plan to make it more effective.

We will add more fields to the existing definition of test suite work items and use them for creating reports. The process of customizing Test Plan and Test Suite is similar to existing work items provided with TFS. Finally we will touch upon the Shared Parameters concepts for Web Testing.

We can achieve the following with this new feature

- The default states for Test Plan can be changed
- Additional fields can be added to the work item
- Custom fields can be added

In this article, we will customize Test Plan to add more transitions, add fields to it, provide rules to the field and use the field to create report. We will need Power Tools to achieve these. I have used a readymade machine from Azure which had Visual Studio 2013 Update 3 and Team Foundation Server 2013 Update 3 installed on it. If you have an Azure subscription, you can do the same or alternatively create a similar setup on your desktop environment.

Open VS > Select Tools > Process Editor > Work Item Types > Open WIT from Server and select Test Plan as work item.

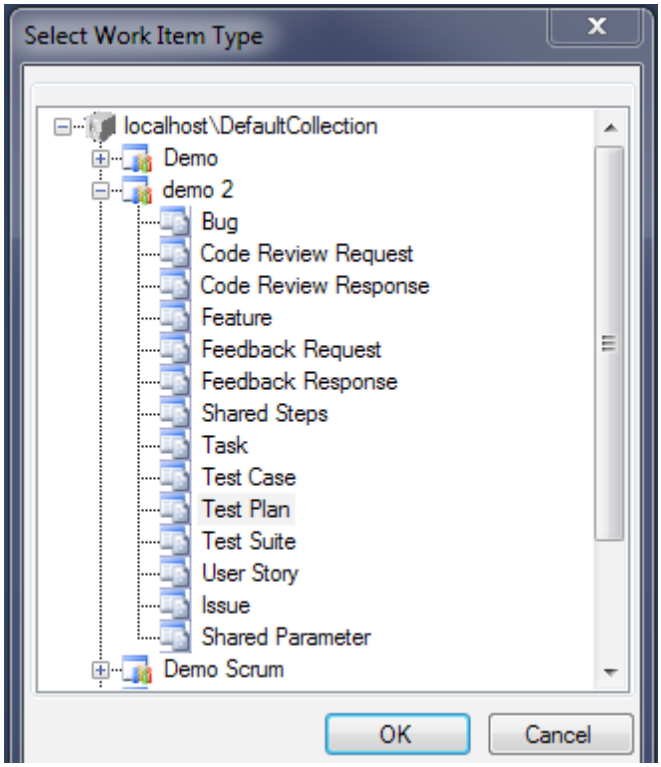


Figure 1: Power Tool Selection for Work Item

We will start with the original workflow of Test Plan which looks as follows:

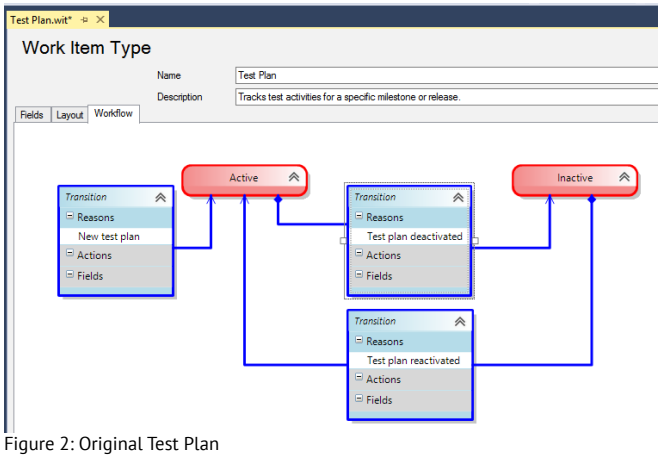


Figure 2: Original Test Plan

I am going to add a few more states to it named Testing, Ready for Sign Off and Completed. Adding a state is in accordance with the work item customization feature we have used previously. Select workflow tab and drag & drop state and transition link wherever required.

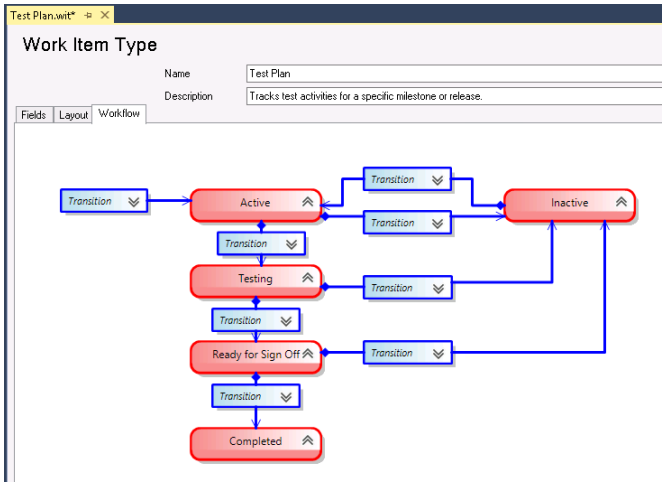


Figure 3: Customized Test Plan

We are going to create a group of Reviewer or Approver. Users in this group will be able to approve or review the Test Plan. We will add a few users in this group.

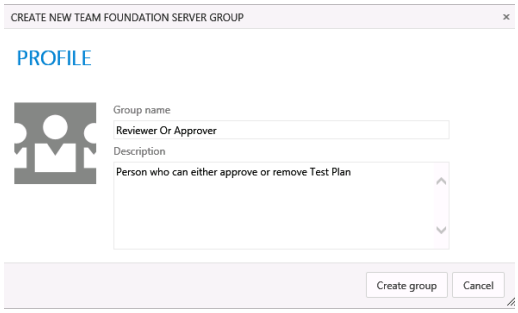


Figure 4: TFS Group

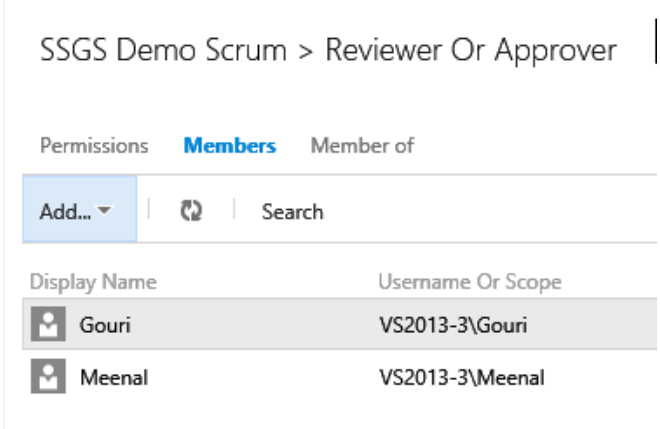


Figure 5: TFS Group Members

We will add a field in the Test Plan named *Reviewer* and provide a rule so that the value for the field can be any user who is member of the newly created Group. A new field created needs a reference name. We want to use this field in the report as a dimension so we will need to define it accordingly.

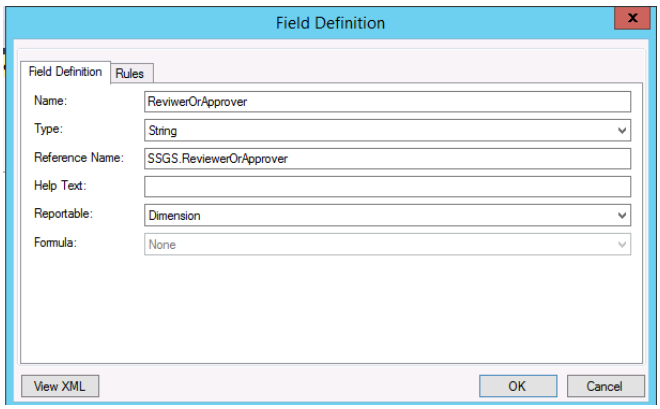


Figure 6: New Field Creation

The Allowed values will take input from the TFS group.

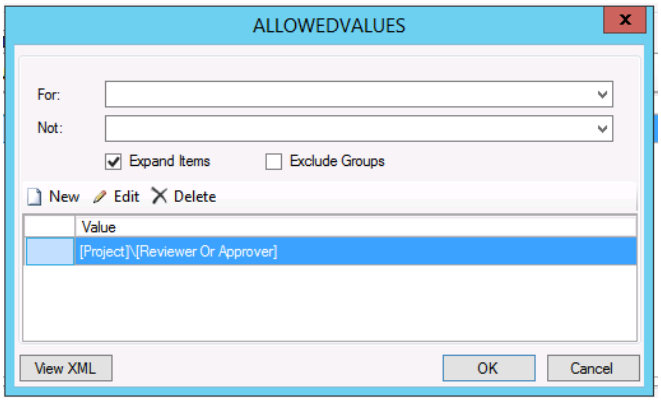


Figure 7: Allowed Values for Field

Now that we have finished with the customization of the Test Plan, let us find out how it looks by using Microsoft Test Manager 2013. I have created a Test Plan and added a couple of Test Suites to it.

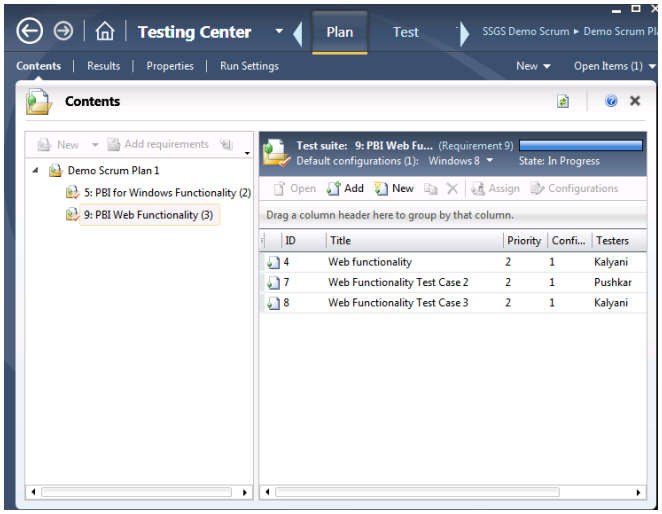


Figure 8: Microsoft Test Manager View

As you can see, first Test Suite has 2 test cases while the second has 3 test cases in it. Let us open the Test Plan and view it.

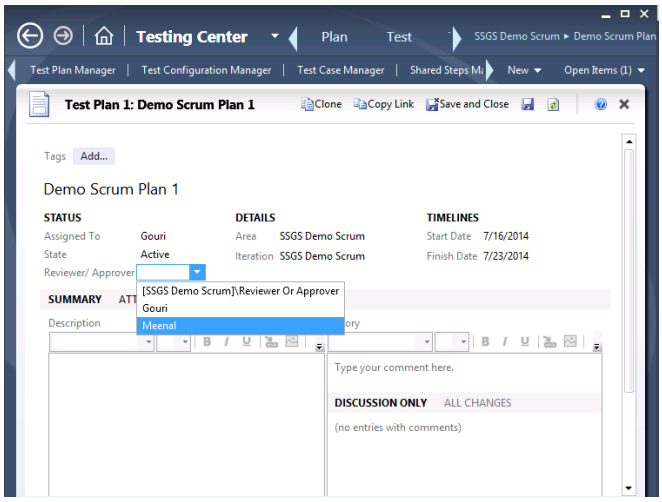


Figure 9: View of Customized Test Plan

Observe that the new field is shown below the field *State*. It also shows the allowed values. In this case, there are 2 users in the Reviewer group who can approve or review the Test Plan.

The current State of the Test Plan is Active. We can change the state to the newly added states and depending upon the transitions we have specified.

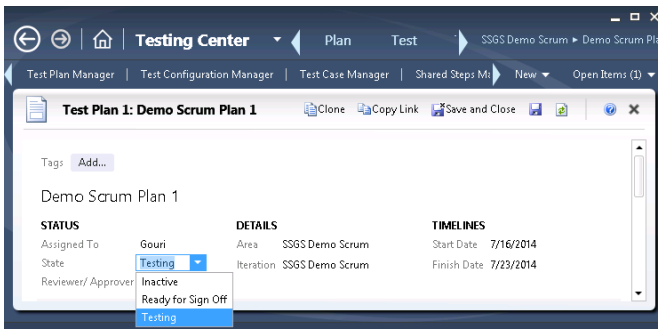


Figure 10: Customized State in Test Plan

The state can change from 'Active' to 'Testing' to 'Ready for Sign Off'. Finally we can change the state to Completed. At any point of time, we can change the state to Inactive saying that this Test Plan is no longer in Active state.

With these kinds of states, we can view the Test Plans in different states. This can be achieved by using a query:

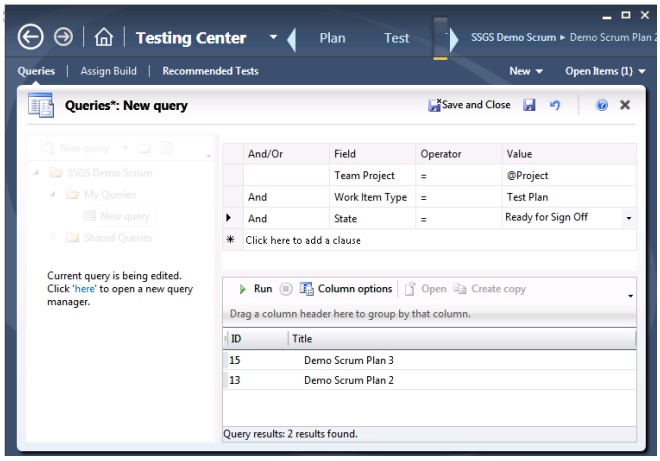


Figure 11: Test Plan Query

The query fetches all the Test Plans which are in 'Ready for Sign Off' State. This query can be stored for further use. We can even create an ad-hoc report using Report Builder for this new functionality.

I have created a report which shows Reviewer wise Test Plan Status. It shows consolidated matrix of various states and their respective reviewers or approvers.

Reviewer wise Test Plan Status

Test Plan State	Gouri	Meenal	Total
Ready for Sign Off	2		
Testing		1	
Total	2	1	

Figure 12: Test Plan Status Report

I have also modified Test Suite to add other Test Types to it.

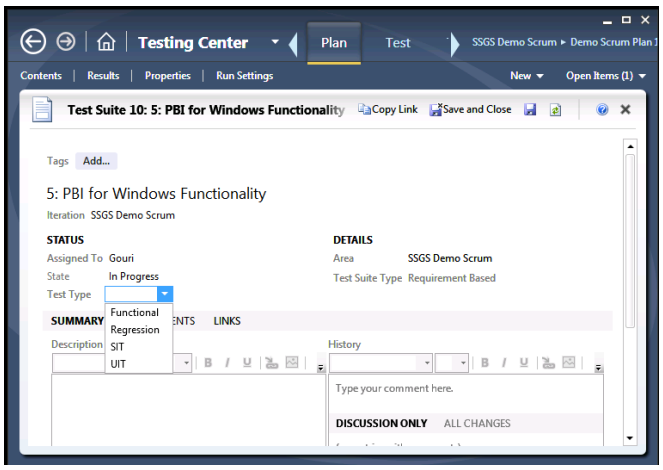


Figure: 13 Customized Test Suite View

As you can see, I have added types like 'Functional', 'Regression, 'System Integration Testing' and 'User Interface Testing'. We can add more types if required like 'UAT'.

I have also created a graphical report which shows the total number of Test Suites in each type.

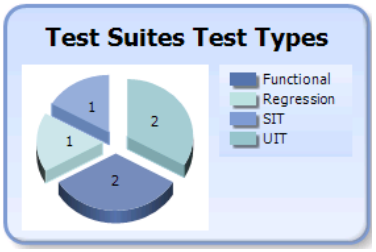


Figure 14: Graphical Report for Test Suites

We have seen so far how the Test Plan and Test Suites are treated as work items and are fully customizable.

I will discuss another feature called *Shared Parameter* which was added in Visual Studio 2013 and Team Foundation Server 2013 Update 2. With Web Testing, you can run Test Cases with multiple sets of data and multiple times. We can add parameter names to the Test Steps and provide the values of the parameters. These shared parameters will be automatically shown in all test cases which reference them. You can either convert existing parameters to shared or create new ones.

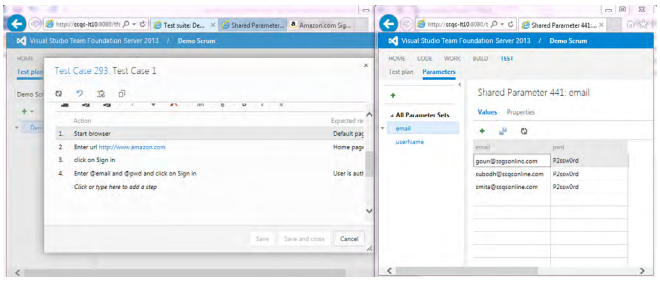


Figure 15: Shared Parameters in Web Testing

We discussed how customization of Test Plan and Test Suites help in creating reports and getting more information via queries. With Test Plan and Test Suite as work items, we can add fields and status of the fields also. We also discussed how Shared Parameters help in Web Testing.

CONCLUSION:

Visual Studio 2013 and Team Foundation Server 2013 Update 3 has made a lot of features available along with providing Test Plan and Test Suites as work items. Customizing work items is a strength of Team Foundation Server which can be leveraged now for Test Plan and Test Suites. Creation of Queries and Custom Reports based upon Test Plan and Test Suite is possible as they are defined as work items ■

Gouri Sohoni, is a Visual Studio ALM MVP and a Microsoft Certified Trainer since 2005. Check out her articles on TFS and VS ALM at bit.ly/dncm-auth-gsoh


Join us on
Facebook
www.facebook.com/
dotnetcurry

Introduction to ServiceStack

ServiceStack is a thoughtfully architected, obscenely fast, thoroughly enjoyable web services for all, built on top of ASP.NET

When we think of building web applications or services in .NET, we usually confine ourselves to core ASP.NET technologies. There is no doubt that Microsoft has put in a lot of efforts in building some awesome web frameworks and tools to make our lives easier, but there are alternatives to Microsoft's set of technologies that are worth taking a look at. ServiceStack is one such option.

ServiceStack is a configuration free, code-first, light-weight framework built on top of ASP.NET for building services and web applications. As the name suggests, it is a stack of services. It provides with just everything that one needs for building end-to-end web services. In a way, we can say that it is a light-weight alternative to WCF, Web API, ASP.NET MVC, ASP.NET Web Forms and any framework using which we can develop web apps or APIs.

WHAT IS THE 'STACK' IN SERVICESTACK?

ServiceStack has a big list of features stacked in, that is just enough to build any kind of service. The following are key components in the stack:

1. REST, SOAP and Message Queuing Services

2. Automatic Serialization/Deserialization to/from a variety of data formats including JSON, XML and CSV (JSON parser is the fastest parser in .NET)

3. A light weight ORM package called OrmLite

4. Dependency Injection

5. Razor views

6. Logging

7. Bundling

8. Security

To learn more about these features, you can visit https://servicestack.net/features. The good thing is, ServiceStack doesn't depend on any third party to support any of its features. It is a self-contained and self-independent framework.

WHY SERVICESTACK?

Using ServiceStack, one can quickly build APIs that can be hosted anywhere (IIS, Windows Service, Self-host or Mono) and consumed from anywhere. Building a ServiceStack API is really easy as it doesn't need a lot of configuration before getting the service up and running. ServiceStack defines a set of conventions that make the job of writing and exposing services easy.

The pieces in the ServiceStack are totally independent of each other and can be used in isolation. For example, if you only want the JSON parser of ServiceStack for your application, you can include it alone and use it.

In addition to all the rich features, ServiceStack embraces the practices of clean code, which means, the logic that we write to expose services using ServiceStack are fully testable.

As listed in the features, ServiceStack supports Razor views. This feature enables to create Razor pages to render the data exposed by the Services.

STUDENT REPORT APPLICATION

In this article, we will build a Student Report application using ServiceStack. By the end of this article, you will be familiar with the following features of ServiceStack:

1. REST Services
2. OrmLite
3. Dependency Injection
4. Razor Views

SETTING UP THE PROJECT

Open Visual Studio 2013 and choose File > New > Project, create an empty ASP.NET web application and name it StudentReports. Install the following NuGet packages in this project:

1. Bootstrap
2. ServiceStack
3. ServiceStack.OrmLite.SqlServer
4. ServiceStack.Razor

To bootstrap ServiceStack, we need to register the ServiceStack components when the application starts. To do this, add a new class to the project and name it *AppHost*. This class should inherit from *ServiceStack.AppHostBase* class.

The class *AppHostBase* doesn't have a non-parameterized constructor. The constructor expects a name for the service and a list of assemblies where the services are defined. As we haven't created services yet, add a class named *StudentService* to the project. This class can be left empty for now. We will come back to this class later to add code to it.

The *AppHostBase* class has an abstract method, Configure which will be overridden in our *AppHost* class. This method is used to configure the components in *ServiceStack*. We can leave it empty for now. Following is the code in *AppHost* as of now:

```
public class AppHost : AppHostBase
{
  public AppHost() : base("Student report Service",
  typeof(StudentService).Assembly) { }

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

And finally to finish the setup, we need to invoke *Init()* method of the *AppHost* in *Application_Start* event of Global.asax. If the project doesn't have a Global.asax file yet, add it and modify the *Application_Start* event:

```
protected void Application_Start(object sender,
EventArgs e)
{
  (new AppHost()).Init();
}
```

Note: *If you want to avoid the above steps while creating a ServiceStack project, install the ServiceStackVS extension through extensions gallery. It installs some project templates that include all the required hooks to build a ServiceStack application.*

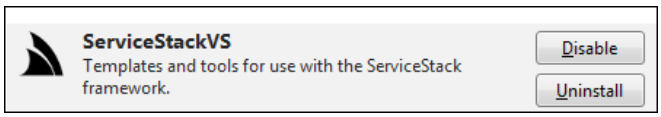


Figure 1: ServiceStackVS extension

WRITING BASIC SERVICES USING SERVICESTACK

ServiceStack is based on DTOs (Data Transfer Object model). To define a service, we need at least one request DTO class. For instance, say we need the ID of a student in the service, then the request DTO can be a class with the field ID. If you need data, additional properties can be added to the DTO class.

We need to expose any service through an endpoint. Endpoints can be created using *Route* attribute defined on *ServiceStack* namespace. Unlike ASP.NET Web API, where we define routes in the service controller, ServiceStack routes are defined on DTOs.

Following is the DTO for student data:

```
[Route("/students")]
[Route("/students/{Id}")]
public class StudentRequestDto
{
  public int Id { get; set; }
}
```

As you see, the above DTO exposes two endpoints: one plain endpoint with no parameters and the second one with Id as the parameter.

You can optionally define response DTO as well. Response DTO can be specified as return type on the request DTO as follows:

```
public class StudentRequestDto :
IReturn<StudentResponseDto>
```

Now let's define a service method to respond to the requests. A service class should be a child to *ServiceStack.Service* class. Open the *StudentService* class created earlier and add the following code to it:

```
public class StudentService : Service
{
  public object Any(StudentRequestDto dto)
  {
    if (dto.Id != default(int))
    {
      return new { Id = dto.Id, Name = "Ravi" };
    }
    else
    {
      return new ArrayList {
      new { Id = 1, Name = "Ravi" },
      new { Id = 2, Name = "Kiran" }
      };
    }
  }
}
```

Names of the methods defined in the service classes have to follow conventions to get some magic applied on them. Here, the method *Any* would respond to any kind of request sent to the endpoints defined over *StudentRequestDto*. Now if you run the application, the following page would appear in the browser:

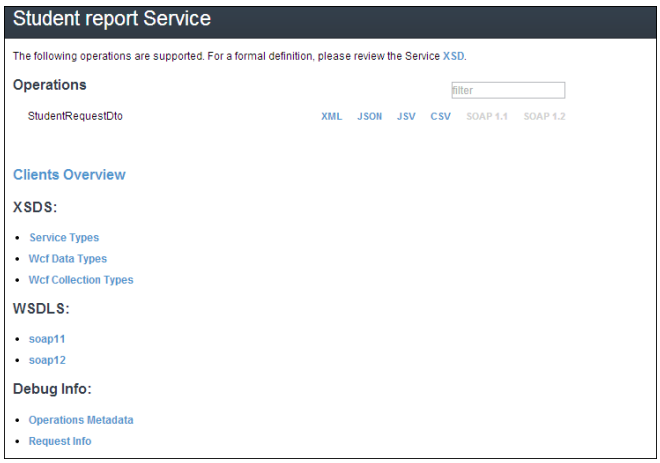


Figure 2: SOAP and REST service

The service by default supports all the formats listed on the page and as we can see, it can be consumed as a SOAP as well as a REST service. The links in the page provide help information about consuming the API. Change the URL to: http://localhost:<port-no>/students?format=json

It would display the data in the form of JSON in the browser:

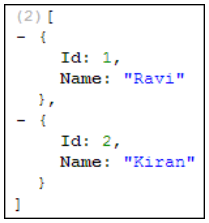


Figure 3: JSON data

You may not like the Pascal Case notation in JSON data. You can modify format of the cases in *Configure* method of the AppHost class. Add the following statement to convert JSON data into camel case:

```
ServiceStack.Text.JsConfig.EmitCamelCaseNames = true;
```

Similarly, change value of format to any of the supported types and you will see the corresponding result.

To restrict a route to HTTP methods, we can specify the list of methods in the Route attribute. Following routes demonstrate 2 cases of restricting:

```
[Route("/students", Verbs = "GET, POST")]  //Only GET
and POST requests are allowed on this route
[Route("/students/{id}", Verbs = "GET")]  //Only GET
request is allowed on this route
public class StudentRequestDto {}
```

In the service class, we need to define the corresponding methods following their conventions.

```
public class StudentService : Service {
  public object Get(StudentRequestDto dto)   {
    //Implementation
  }

  public object Post(StudentRequestDto dto)
  {
    /Implementation
  }
}
```

For the demo Student Reports application, we don't need the POST event on Student. But we will be performing GET, POST and PUT operations on Marks. Following is the Marks DTO with required Routes:

```
[Route("/marks/student/{StudentId}", Verbs = "GET")]
[Route("/marks/{MarksId}", Verbs = "GET, PUT")]
[Route("/marks",Verbs="POST")]
public class MarksRequestDto {
    public int MarksId { get; set; }
    public int StudentId { get; set; }
    public int MarksAwarded { get; set; }
    public int MaxMarks { get; set; }
    public string Subject { get; set; }
}
```

Notice the first route in the above snippet. This route is to enable fetching marks of a particular student. The route is defined this way to avoid confusion between accepting MarksId and StudentId into the API. We will define a service for this DTO later.

USING ORMLITE AND DEPENDENCY INJECTION

ServiceStack includes a light-weight micro ORM called OrmLite. It is one of the fastest .NET ORMs. It is not as configurable as Entity Framework, but still provides a good set of features to interact with databases. OrmLite has DB specific implementations for all major databases including SQL Server, Oracle and MySql and some others like PostgreSQL, FirebirdSql

and Sqlite. We will be using the SQL Server version.

As a first step, create a new database on SQL Server and name it StudentDb. Add the following connection string to the Web.config:

```
<add name="studentDbConn" connectionString="Data
Source=.; Initial Catalog=StudentDb; Integrated
Security=true;" />
```

To work with the database, we need an object of *IDbConnectionFactory* type. *OrmLiteConnectionFactory* is an implementation of this interface. Following statement creates an object of this class:

```
var ormLiteConnectionFactory = new
OrmLiteConnectionFactory(ConfigurationManager.
ConnectionStrings["studentDbConn"].
ConnectionString,  ServiceStack.OrmLite.SqlServer.
SqlServerOrmLiteDialectProvider.Instance);
```

Let's make this object available to the entire application through the built-in IoC container of ServiceStack *Funq*. Funq in ServiceStack is based on an open-source IoC container named Funq (http://funq.codeplex.com/) with the framework adding more capabilities to its implementation. ServiceStack internally uses Funq to register objects of the Service classes, Filters and Validators.

Let's register the above object to the container of Funq. *Configure* method of the *AppHost* class already has a reference to Funq's container. Following statement registers the dependency:

```
container.Register<IDbConnectionFactory>
(ormLiteConnectionFactory);
```

Objects registered to Funq are injected through both constructor and property injection.
To read more about Funq, visit ServiceStack's wiki on GitHub (http://bit.ly/dncm15-sswiki).

Let's set up the database with two tables to store Student data and marks and fill-in some data into the tables. Following are the classes for Student and Marks tables:

```
public class Student
{
  [AutoIncrement]
  public int StudentId { get; set; }
  public string Name { get; set; }
  public string City { get; set; }
  public int CurrentClass { get; set; }
}
```

The *AutoIncrement* attribute in above snippets denotes a table column with auto incrementing values, which is Identity column in case of SQL Server. References attribute in the Marks class defines a foreign key reference.

To fill these tables with data, let's create a new class *DbInitializer*. This class will have just one static method *InitializeDb*. This method checks for existence of tables, creates the tables and fills in data if the tables don't exist yet.

```
public class DbInitializer
{
  public static void InitializeDb(IDbConnectionFactory
  dbConnectionFactory)
  {
    var students = new List<Student>()
    {
      new Student()
      {Name="Andrew",City="Boston",CurrentClass=2},
      new Student()
      {Name="Richa",City="Chicago",CurrentClass=3},
      new Student()
      {Name="Dave",City="Phoenix",CurrentClass=4},
      new Student()
      {Name="Ema",City="Washington",CurrentClass=5},
      new Student()
      {Name="Filip",City="Texas",CurrentClass=6},
      new Student(){Name="Maggi",City="Los
      Angeles",CurrentClass=7},
      new Student()
      {Name="Nathan",City="Atlanta",CurrentClass=8}
    };

    var studentMarks = new List<Marks>()
    {
      new Marks() {Subject="Mathematics",MarksAwarded=80,
      MaxMarks=100, StudentId=1},
      new Marks() {Subject="English",MarksAwarded=70,
```

```
      MaxMarks=100, StudentId=1},
      new Marks() {Subject="Hindi",MarksAwarded=75,
      MaxMarks=100, StudentId=1},
      new Marks(){Subject="Mathematics",MarksAwarded=60,
      MaxMarks=100, StudentId=2},
      new Marks() {Subject="English",MarksAwarded=90,
      MaxMarks=100, StudentId=3},
      new Marks(){Subject="Hindi",MarksAwarded=85,
      MaxMarks=100, StudentId=2},
      new Marks(){Subject="Mathematics",MarksAwarded=90,
      MaxMarks=100, StudentId=2},
      new Marks(){Subject="English",MarksAwarded=80,
      MaxMarks=100, StudentId=3},
      new Marks(){Subject="Hindi",MarksAwarded=80,
      MaxMarks=100, StudentId=3}
    };

    using (var db = dbConnectionFactory.
    OpenDbConnection())
    {
      if (!db.TableExists("Student"))
      {
        db.CreateTable<Student>();
        db.InsertAll<Student>(students);
      }

      if (!db.TableExists("Marks"))
      {
        db.CreateTable<Marks>();
        db.InsertAll<Marks>(studentMarks);
      }
    }
  }
}
```

To operate with these tables, we need repositories. As we have just two tables and we need limited number of operations to be performed on the tables, one repository class will suffice. The repository class would be performing the following operations:

- Get all students
- Get student details by id
- Get marks of a student
- Get marks by marks ID
- Add marks of a student
- Update marks of a student

We need a connection factory object to establish a connection to the database. As the object is already added to Funq, we will get it through property injection. Following is the repository class with a method that fetches all students from the DB:

```
public class StudentDbRepository
{
  public IDbConnectionFactory DbConnectionFactory {
  get; set; }

  public List<Student> GetStudents()
  {
    using (var db = DbConnectionFactory.
    OpenDbConnection())
    {
      return db.Select<Student>();
    }
  }
}
```

In the above snippet, *Select()* is a generic method that gets data from the table represented by the class specified with it. We enclosed the data operation inside a *using* block to dispose the connection object immediately after the operation is done. To apply filter while fetching data, we can pass a lambda expression to the *Select()* method as shown in the following methods:

```
public List<Marks> GetMarksByStudent(int studentId)
{
  using (var db = DbConnectionFactory.
  OpenDbConnection())
  {
    return db.Select<Marks>(m => m.StudentId ==
    studentId);
  }
}

public Marks GetMarks(int marksId)
{
  using (var db = DbConnectionFactory.
  OpenDbConnection())
  {
    return db.Select<Marks>(m => m.Id == marksId).
    FirstOrDefault();
  }
}
```

To add marks of a student, we can call the *Insert()* method on the connection object. If the *Insert()* method adds data to a table with an identity column in it, the identity value can be fetched using *LastInsertedId()* method on the same connection object. Following snippet demonstrates this while adding a new entry of marks:

```
public int AddMarks(Marks marks)
{
  using (var db = DbConnectionFactory.
  OpenDbConnection())
  {
    db.Insert(marks);
    return (int)db.LastInsertId();
  }
}
```

The last operation that we need to perform is updating marks. It is a straight forward operation and returns the number of rows affected.

```
public int UpdateMarks(Marks marks)
{
  using (var db = DbConnectionFactory.
  OpenDbConnection())
  {
    return db.Update(marks);
  }
}
```

With this, the repository is completed and now we can use it in our services. As the service classes need an instance of the repository class, let's make the instance available through the dependency injector Funq.

```
container.Register(c => new StudentDbRepository()).
ReusedWithin(ReuseScope.Request);
```

The *ReuseWithin()* method chained with the *Register()* method is used to define scope of an instance registered. The repository object is scoped within the HTTP request. The object is destroyed as soon as the request ends.

Following are the modified *StudentService* and *MarksService* classes.

```
public class StudentService : Service {
  StudentDbRepository repository;
  public StudentService(StudentDbRepository _repository)
  {
    repository = _repository;
  }

  public object Get(StudentRequestDto studentDto)
  {
    if (studentDto.Id == default(int))
    {
      return repository.GetStudents();
    }
    else
    {
      return repository.GetStudentById(studentDto.Id);
    }
  }
}

public class MarksService : Service
{
  StudentDbRepository repository;

  public MarksService(StudentDbRepository _repository)
  {
    repository = _repository;
  }

  public object Get(MarksRequestDto dto)
  {
    if (dto.StudentId != default(int))
    {
      var student = repository.GetStudentById(dto.
      StudentId);
      var marks = repository.GetMarksByStudent(dto.
      StudentId);

      return new MarksGetResponseDto()
      {
        Id = student.StudentId,
        Name = student.Name,
        Class = student.CurrentClass,
        Marks = marks
      };
    }
    else if (dto.MarksId != default(int))
    {
```

```
      var marks = repository.GetMarks(dto.MarksId);
      var student = repository.GetStudentById(marks.
      StudentId);
      return new MarksGetResponseDto()
      {
        Id = student.StudentId,
        Name = student.Name,
        Class = student.CurrentClass,
        Marks = new List<Marks>() { marks }
      };
    }
    return null;
  }

  public object Post(MarksRequestDto dto)
  {
    var nextId = StaticStudentDb.
    studentMarks[StaticStudentDb.studentMarks.Count()
    - 1].Id;
    var newStudentMarks = new Marks()
    {
      StudentId = dto.StudentId,
      MarksAwarded = dto.MarksAwarded,
      MaxMarks = dto.MaxMarks,
      Subject = dto.Subject
    };

    var id = repository.AddMarks(newStudentMarks);
    newStudentMarks.Id = id;

    return newStudentMarks;
  }

  public object Put(MarksRequestDto dto)
  {
    return repository.UpdateMarks(new Marks()
    {
      Id = dto.MarksId,
      Subject = dto.Subject,
      StudentId = dto.StudentId,
      MarksAwarded = dto.MarksAwarded,
      MaxMarks = dto.MaxMarks
    });
  }
}
```

Here is the *MarksGetResponseDto* used in the above snippet:

```
public class MarksGetResponseDto
{
  public int Id { get; set; }
  public string Name { get; set; }
  public int Class { get; set; }
  public IEnumerable<Marks> Marks { get; set; }
}
```

USING HTTP RESPONSE

In the services we wrote till now, we are returning plain objects. ServiceStack internally wraps this data inside an HTTP response before sending it to the user. But you may want to get control over the HTTP response and send data along with a status code of your choice to the client. This can be done using *IHttpResult*.

Let's refactor the Get method of the *StudentService* to use *IHttpResult*:

```
public IHttpResult Get(StudentRequestDto studentDto)
{
  if (studentDto.Id == default(int))
  {
    var result = new HttpResult(repository.
    GetStudents());
    return result;
  }
  else
  {
    var student = repository.GetStudentById(studentDto.
    Id);
    if (student != null)
    {
      return new HttpResult(student);
    }
    else
    {
      return new HttpError(HttpStatusCode.
      NotFound,"Student with id "+studentDto.Id
      + " doesn't exist.");
    }
  }
}
```

Both *HttpResult* and *HttpResult* have a number of overloaded constructors offering flexibility to set status code, response type, message and a number of other HTTP parameters of our

choice.

RAZOR VIEWS

Till now, we used the Service and Data access features of ServiceStack. Let's complete the demo application by adding some views. ServiceStack supports Razor and markdown razor views. If you are already familiar with the Razor views in ASP.NET MVC, you don't need to learn anything new to use the view engine of ServiceStack.

The difference between ServiceStack and MVC appears when we see the way views are rendered in these frameworks. Unlike ASP.NET MVC, ServiceStack doesn't need a controller. Data for the view is passed from the corresponding Service method. This data is available to the view in the Model object and the view can use it for model binding. Views are rendered on the same endpoint where the service is exposed. But we can still get the data served from these endpoints by specifying format of the data in the URL.

We have already added assembly required for Razor views while installing the NuGet packages. We can start building the views now. As a first step, let's configure Razor as a plugin on start of the application.

```
Plugins.Add(new RazorFormat());
```

By default, an application doesn't know about the razor views. We must add some configurations in Web.config to be able to use the view engine seamlessly. Luckily, the NuGet package ServiceStack adds all the required configurations to Web.config. You can check your Web.config file to see the new configurations.

By convention, ServiceStack looks for its views under a folder named Views. So create a new folder and change its name to Views. To keep the UI consistent, let's create a layout file. Add a new file to the Views folder and name it _Layout.cshtml. Add the following code to this file:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>@ViewBag.Title</title>
  <link href="~/Content/bootstrap.min.css"
  rel="stylesheet" />
</head>
<body>
  <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">
    <div class="container" style="margin-top:100px;">
      @RenderBody()
    </div>
  </section>
  <script src="~/Scripts/jquery-1.9.0.min.js"></script>
  @RenderSection("Scripts", required: false)
</body>
</html>
```

Like in case of MVC, the views in ServiceStack get the dynamic object ViewBag. Here we are using it to set title of the page dynamically.

By convention, names of the Views should be same as the name of the corresponding request DTO classes. As we have two DTO classes, add two files named StudentRequestDto.cshtml and MarksRequestDto.cshtml to the Views folder. Both these files automatically extend the layout file, we don't need to declare it anywhere.

Razor mark-up in the StudentRequestDto.cshtml view is going to be very simple, as it has to loop through the student objects and display their details along with a link to the marks page. Following is the Razor mark-up in this view:

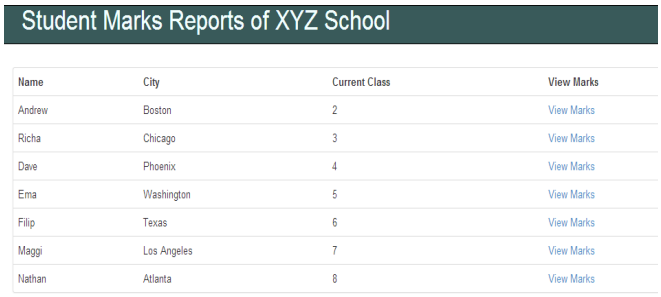
```
@using StudentReports.Models
@inherits ViewPage<IEnumerable<Student>>

@{
    ViewBag.Title = "Student List";
}

<div class="row">
  <div class="panel panel-default">
    <table class="table">
      <thead>
        <tr>
```

```
            <th>Name</th>
            <th>City</th>
            <th>Current Class</th>
            <th>View Marks</th>
          </tr>
        </thead>
        <tbody>
        @{
            foreach (var item in Model)
            {
              <tr>
                <td>@item.Name</td>
                <td>@item.City</td>
                <td>@item.CurrentClass</td>
                <td>
                  <a href="/marks/student/@item.
                  StudentId">View Marks</a>
                </td>
              </tr>
            }
          }
        </tbody>
      </table>
    </div>
</div>
```

Run the application and change the address in the browser to: http://localhost:<port-no>/students
It should display the following page:



The marks page is going to be more functional than the Students list page as it has to perform fetch, add and update marks operations. On load, the page gets list of marks of the student from Get method of MarksService and this data is available in the Model object. For adding and updating marks, we will invoke the corresponding REST APIs using jQuery AJAX.

Following is the mark-up in the page:

```
@using StudentReports.Models
@inherits ViewPage<StudentReports.DTOs.
MarksGetResponseDto>

@{
  ViewBag.Title = "Marks of " + Model.Name + " studying
  in class " + Model.Class;
}

<div class="row">
  <div class="text-center">
    <div class="row">
      <div class="col-md-3"></div>
      <div class="col-md-3"><b>Name: </b></div>
      <div class="col-md-3">
        @Model.Name
        <input type="hidden" value="@Model.Id"
        id="hnStudentId" />
      </div>
      <div class="col-md-3"></div>
    </div>
    <div class="row">
      <div class="col-md-3"></div>
      <div class="col-md-3"><b>Class: </b></div>
      <div class="col-md-3">
        @Model.Class
      </div>
      <div class="col-md-3"></div>
    </div>
  </div>
  @{
    if (Model.Marks == null || Model.Marks.Count() ==
    0)
    {
      <div class="info">No marks added for the student
      yet.</div>
    }
    else
    {
      <table class="table" id="tblStudentMarks">
        <thead>
          <tr>
            <th>Subject</th>
            <th>Marks Awarded</th>
            <th>Max Marks</th>
            <th>Update</th>
          </tr>
```

```
        </thead>
        <tbody>
          @{
            foreach (var item in Model.Marks)
            {
              <tr>
                <td>@item.Subject</td>
                <td>@item.MarksAwarded</td>
                <td>@item.MaxMarks</td>
                <td><button class="btn btn-link" data-
                  marks-id="@item.Id">Update</button>
                </td>
              </tr>
            }
          }
        </tbody>
      </table>
    }
  }
  <div class="pull-right">
    <button id="btnAddMarks" class="btn btn-
    default">Add Marks</button>
  </div>
  <div id="divNewMarks" style="display: none;">
    <form id="formNewMarks">
      <input type="hidden" value="@Model.Id"
      name="studentId" />
      <div class="row">
        <div class="col-md-3"></div>
        <div class="col-md-3">Subject</div>
        <div class="col-md-3">
          <input name="subject" type="text"
          id="txtNewSubject" />
        </div>
        <div class="col-md-3"></div>
      </div>
      ...
      ...
      </div>
    </form>
  </div>
</div>
```

Note: *The markup has been truncated here to save some space. Please download the source code to view the complete markup.*

In the layout page, we have defined an optional script section that can be used in the child pages to include custom scripts. Before executing the custom script, we would have all the libraries loaded.

Let's write some code to add new marks for a student. By default, the add marks form is invisible to the user. On click of the Add Marks button, we need to show the form and accept inputs. When the data is successfully posted to the server, we refresh the page to see all the marks of the student. Following is the script for adding new marks within the scripts section:

```
@section Scripts{
  <script>
    $(function () {
      $("#btnAddMarks").click(function () {
        $("#divNewMarks").show();
      });

      $("#btnAddNewMarks").click(function () {
        var newMarks = $("#formNewMarks").serialize();

        $.post("/marks?format=json", newMarks)
        .then(function (result) {
          window.location.reload();
        });
      });
    });
  </script>
}
```

Alternatively, we can bind data to the table manually to avoid refreshing the page. I used this approach to keep things simple. Similarly, we need to write script to send a PUT request when marks are modified. I am leaving this part as an assignment. The solution is available in the downloadable code for this article.

CONCLUSION

I hope this article helped you to get started with ServiceStack and also got your hands dirty with it. Using ServiceStack, it is easier to build services and views and get them running on any platform of your choice. The good thing is, it doesn't depend on any other third parties to make any of its component work. At the same time, the components in the framework are built with a vision of easier usability, speed and modularity in mind.

ServiceStack is not limited to the set of features discussed in this article. We will have some more articles in future focusing on more capabilities of this great framework. I encourage you to check out the GitHub projects (github.com/ServiceStack/) and their official wiki (https://github.com/ServiceStack/ServiceStack/wiki) to learn more about the framework ■

Download the entire source code from our GitHub Repository at bit.ly/dncm15-servicestack

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

AZURE BLOB STORAGE SNAPSHOTS USING CLIENT LIBRARY AND REST APIS

A Blob storage can be thought of as a special table storage in the cloud. Azure Blob storage is a service for storing large amounts of data in binary format as data chunks that can be accessed via HTTP or HTTPS. One of the coolest features of the Azure blob storage is the ability to create snapshots of a blob file. The snapshot allows you to preserve the blob as of a specific point in time.

This article demonstrates various operations that can be used to create and delete snapshots as well as restore blob from snapshots using client library and REST APIs.

INTRODUCTION

Azure Storage is a very widely used service in most Azure based applications. I personally have not seen a single project running on Azure and not using the Azure Storage. Azure Storage has four main offerings –

1. Blob
2. Table
3. Queue
4. Drive – Read *Drives* as *Azure Files* now, as Drives will be deprecated in the year 2015.

Out of these offerings, this article will focus on the snapshot feature of Azure Blob storage and how it can be effectively used in azure based applications. This article assumes that you have a basic background of Azure Blob storage and understand its primary use.

If you refer to the dictionary meaning of Snapshot, it states that – *a photograph, taken with a handheld camera with no specific aim.* To a developer's delight, we can take snapshot of a Azure blob storage. Of course not with the Camera, but it serves a similar purpose i.e. capture an instance at a specific point in time. Blob snapshots have a specific aim. This aim is of improving performance and maintaining backup or original copy of the blob present in azure storage.

EXPLORING THE AZURE BLOB SNAPSHOT

Snapshot offers you a read-only copy of the blob storage. As it is read-only, you can read, copy and delete, but cannot edit/modify. Blob snapshots are very fruitful in case of Azure Virtual Machines. For example, let's say you are configuring non Microsoft software on Azure Virtual Machines and it involves minimum 50 to 60 steps to complete the configuration. You also need to make sure that after each step performed, VM is in a running condition and there are no errors. If at all an error occurs, you want to revert back to the previous working/running condition of VM. Such scenarios are ideal for the use of Azure blob Storage snapshot. This is very similar to Team Foundation Server (TFS) capability where you maintain the versions of your code. Here you maintain different versions of blob differentiated by Date, on which the snapshot is taken.

CREATE SNAPSHOT FOR A BLOB

The following steps will explain how you can create snapshot for a blob. For demo purposes, I am using Azure Storage Emulator. The Microsoft Azure storage emulator provides a local environment that emulates the Azure Blob, Queue, and Table services and allows you test your application without paying any money. Before creating a snapshot, it is important to upload a blob to Azure storage first. I used the standard code available here (http://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-how-to-use-blobs/) to perform the upload to azure blob storage. I just made one change in this code. While uploading the content to blob storage, I also added property and metadata to the block blob with following code:

```
blockBlob.Properties.CacheControl = "max-age=60,
must-revalidate";
blockBlob.SetProperties();
blockBlob.Metadata.Add("author", "kunal");
```

Now I opened the server explorer from Visual Studio 2013 and expanded the storage node for development environment, where I can see the blob uploaded with added property information, as shown in the screenshot below

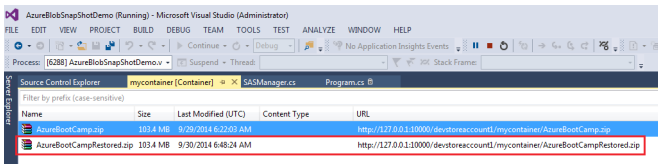


Figure 1: Blog Property Information

Let us see how to programmatically create Blog snapshot using Client Library and REST API:

A.USING AZURE CLIENT LIBRARY –

The following method illustrates the create snapshot code using Azure storage client library -

```
private static void CreateSnapshotUsingClientLibrary(){
  //specifying container name and blob name - change
  //them as per your blob and container name
  string containerName = "mycontainer";
  string blobName = "AzureBootCamp.zip";
  // Retrieve storage account from connection string
  CloudStorageAccount storageAccount =
```

```
  CloudStorageAccount.Parse(storageConnectionString);

  // Create the blob client
  CloudBlobClient blobClient = storageAccount.
  CreateCloudBlobClient();

  // Get a reference to the container  and blob
  CloudBlobContainer container = blobClient.
  GetContainerReference(containerName);
  CloudBlockBlob blob = container.
  GetBlockBlobReference(blobName);

  //create snapshot
  CloudBlockBlob snapshot = blob.CreateSnapshot();

  //list down the Uri of snapshot created
  Console.WriteLine("SnapshotQualifiedUri: " + snapshot.
  SnapshotQualifiedUri);
  Console.WriteLine("Snap shot time:" + snapshot.
  SnapshotTime);
}
```

The Output window will display the URL of the snapshot taken. The format of the snapshot url is as shown here:

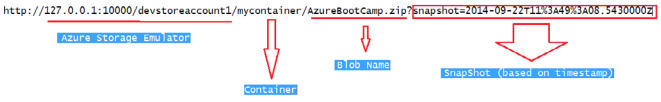


Figure 2: Snapshot url

B. USING REST API

For REST API, it is mandatory to provide the Authorization, Date and x-ms-version information during snapshot creation. For authorization, we will use Shared Access Signature(SAS) created on blob storage. The code to generate SAS based on the stored access policy is pretty straight forward and can be obtained from the link –

http://sanganakauthority.blogspot.com/2012/05/windows-azure-shared-access-signature.html

In this sample, we are using similar code to generate Ad-Hoc type of SAS. To create snapshot using REST API, you need to provide query string parameter as – `&comp=snapshot`

The complete method to create a snapshot is as follows –

```
private static void CreateSnapshotUsingREST() {
  try  {
    //specifying container name and blob name - change
    //them as per your blob and container name
    string containerName = "mycontainer";
    string blobName = "AzureBootCamp.zip";
    string contentType = string.Empty;
    string snapshotTime = string.Empty;
    DateTime now = DateTime.UtcNow;

    //to perform any operation first lets generate the
    //SAS url on the container, validity 1 minute
    SASManager sasMgr = new
    SASManager(storageConnectionString);
    string sasUrl = sasMgr.
    GetAdHocWriteOperationSAS(containerName, blobName);

    //perform operation to create snapshot
    HttpWebRequest requestCreateSnapshot =
    (HttpWebRequest)WebRequest.Create(sasUrl +
    "&comp=snapshot");
    requestCreateSnapshot.ContentLength = 0;
    requestCreateSnapshot.Headers.Add("x-ms-version",
    "2014-02-14");
    requestCreateSnapshot.Headers.Add("x-ms-date",
    now.ToString("R", System.Globalization.CultureInfo.
    InvariantCulture));
    requestCreateSnapshot.Method = "PUT";

    using (HttpWebResponse respCreateSnapshot =
    (HttpWebResponse)requestCreateSnapshot.GetResponse())
    {
      //create operation returns CREATED response
      if (respCreateSnapshot.StatusCode ==
      HttpStatusCode.Created) {
        if (respCreateSnapshot.Headers != null) {
          snapshotTime = respCreateSnapshot.Headers.
          Get("x-ms-snapshot");
        }
      }
    }
    Console.WriteLine("snapshot time - " +
    snapshotTime);
  }
  catch (Exception ex) { throw ex; }
}
```

The snapshot time is the time value at which snapshot got created and unique identifier for your snapshot. So this parameter will be applied at the end of SAS url. Therefore full url to create snapshot of the blob will be as follows –

http://127.0.0.1:10000/devstoreaccount1/mycontainer/AzureBootCamp.zip?snapshot=2014-09-25T10:38:31.5670000Z

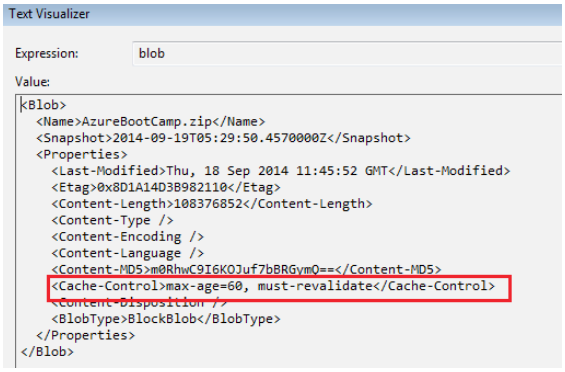


Figure 3: Snapshot properties and metadata

‘

If you see Figure 3; you can observe that snapshot has all the properties/ metadata exactly same as original blob.

LISTING BLOB SNAPSHOTS

The snapshot is distinguished by its time stamp. The time at which snapshot is taken is simply appended as query string to existing URL of the blob storage and that forms the complete url for snapshot of the blob. The same property can be used to list down snapshots. Let’s see how to do it using Client Library and REST API.

A. USING CLIENT LIBRARY –

```
private static void ListSnapshotsForBlob()
{
  //specifying container name and blob name - change
  //them as per your blob and container name
  string containerName = “mycontainer”;
  string blobName = “AzureBootCamp.zip”;

  // Retrieve storage account from connection string
  CloudStorageAccount storageAccount =
  CloudStorageAccount.Parse(storageConnectionString);

  // Create the blob client
  CloudBlobClient blobClient = storageAccount.
  CreateCloudBlobClient();
  // Get a reference to the container  and blob
  CloudBlobContainer container = blobClient.
  GetContainerReference(containerName);
  CloudBlockBlob blob = container.
  GetBlockBlobReference(blobName);

  //retrive list of snapshots
  IList<IListBlobItem> snapshots = container.
  ListBlobs(null, true, BlobListingDetails.Snapshots).
  Where(x => ((CloudBlockBlob)x).IsSnapshot).ToList();

  //write total number of snapshots for blob
  Console.WriteLine(“Total snapshots:” + snapshots.
  Count + Environment.NewLine);

  foreach (IListBlobItem snapshot in snapshots)
  {
    Console.WriteLine(“Snapshot Uri:” +
    ((CloudBlockBlob)snapshot).SnapshotQualifiedUri
    + Environment.NewLine + “Snapshot Timestamp:” +
    ((CloudBlockBlob)snapshot).SnapshotTime);
  }
}
```

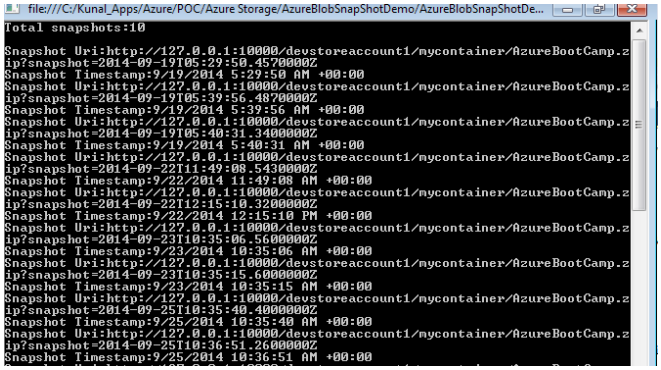


Figure 4: Snapshot created on various timestamp

The following screenshot shows the snapshot created on various timestamp for the blob.

B. USING REST API

```
private static void ListSnapshotsUsingREST()
{
  try
  {
```

```
//specifying container name and blob name - change
//them as per your blob and container name
string containerName = "mycontainer";
List<string> snapshots = new List<string>();

//to perform any operation first let's generate the
//SAS url on the container, validity 1 minute
SASManager sasMgr = new
SASManager(storageConnectionString);
string sasUrl = sasMgr.
GetAdHocListSnapshotsSAS(containerName);

//perform operation to create snapshot
HttpWebRequest request = (HttpWebRequest)
WebRequest.Create(sasUrl + "&restype=container&comp=
list&include=snapshots");
request.ContentLength = 0;
request.Headers.Add("x-ms-version", "2014-02-14");
request.Method = "GET";

using (HttpWebResponse response = (HttpWebResponse)
request.GetResponse())
{
  if (response.StatusCode == HttpStatusCode.OK)//
  list operation returns OK response
  {
    using (StreamReader reader = new
    StreamReader(response.GetResponseStream()))
    {
      string result = reader.ReadToEnd();

      //read snapshot from xml
      XElement x = XElement.Parse(result);
      foreach (XElement blob in x.Element("Blobs").
      Elements("Blob"))
      {
        if (blob.Element("Snapshot") != null)
        {
          snapshots.Add(blob.Element("Snapshot").
          Value);
        }
      }
    }
  }
}
//print snapshots name
foreach (string snapshot in snapshots)
  {
    Console.WriteLine("Snapshot Name:" + snapshot);
  }
 }
 catch (Exception ex)
 {
  throw ex;
 }
}
```

RESTORE BLOB FROM SNAPSHOT

You can restore to the previous version of the blob storage using Snapshots. This is the main purpose of blob snapshot. The following code illustrates how you can restore blob from snapshot. For restore purposes, I am retrieving the first snapshot and using its url to perform restore operation.

A. USING CLIENT LIBRARY

```
private static void RestoreFromSnapshot()
{
  //specifying container name and destination blob
  //name - change them as per your destination blob and
  //container name
  string containerName = "mycontainer";
  string destinationBlobName = "AzureBootCampRestored.
  zip";

  // Retrieve storage account from connection string
  CloudStorageAccount storageAccount =
  CloudStorageAccount.Parse(storageConnectionString);

  // Create the blob client
  CloudBlobClient blobClient = storageAccount.
  CreateCloudBlobClient();

  // Get a reference to the container and blob
  CloudBlobContainer container = blobClient.
  GetContainerReference(containerName);
  CloudBlockBlob destinationBlob = container.
  GetBlockBlobReference(destinationBlobName);

  /*retrive the snapshot url from the first snapshot
  - here you can retrive or use snapshot url of your
  choice. For demo purpose I am considering the first
  snapshot here.*/
```

```
  string snapshotUri = ((CloudBlockBlob)(container.
  ListBlobs(null, true, BlobListingDetails.Snapshots).
  Where(x => ((CloudBlockBlob)x).IsSnapshot).
  FirstOrDefault())).SnapshotQualifiedUri.AbsoluteUri;
  /*or you can specify the snapshot uri of your choice
  as below */
  string snapshotUri = "http://127.0.0.1:10000/
  devstoreaccount1/mycontainer/AzureBootCamp.
  zip?snapshot=2014-09-19T05:29:50.4570000Z";

  //perform copy/restore operation from snapshot uri
  string taskId = destinationBlob.StartCopyFromBlob(new
  Uri(snapshotUri));

  Console.WriteLine("Restore blob url task Id:" +
  taskId);

  while (destinationBlob.CopyState.Status ==
  CopyStatus.Pending)
  {
    Task.Delay(TimeSpan.FromSeconds(20d)).Wait();
    destinationBlob = (CloudBlockBlob)container.
    GetBlobReferenceFromServer(destinationBlobName);
  }
  Console.WriteLine("Copy operation complete");
}
```

After successful restore operation, a new blob gets created having same properties/ metadata as that of snapshot as shown in the image

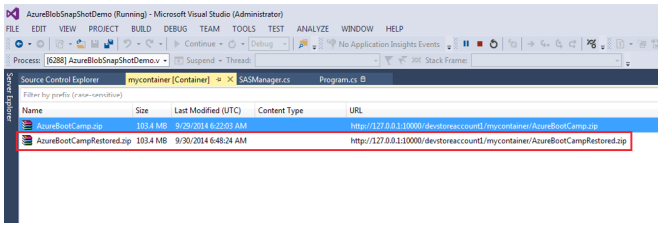


Figure 5: New Blob with same properties as that of snapshot

B. USING REST API

In case of a REST based example, I am using the hardcoded snapshot uri parameter and it will change as per your snapshot.

```
private static void RestoreFromSnapshotUsingREST()
{
  try
  {
    //specifying container name and blob name - change
    //them as per your blob and container name
    string containerName = "mycontainer";
     //my source and destination containers are same.
    string blobName = "AzureBootCamp.zip";
    string destinationBlobName =
          "AzureBootCampRestored.zip";
    string copyStatus = string.Empty;

    //to perform any operation first, let's generate the
    //SAS url on the source container-blob, validity 1
    //minute
    SASManager sasMgr = new SASManager
    (storageConnectionString);
    string sasUrl = sasMgr.
    GetAdHocWriteOperationSAS(containerName, blobName);

    //create sas on destination container-blob
    string destinationSASUrl = sasMgr.
    GetAdHocWriteOperationSAS(containerName,
    destinationBlobName);

    //create request for destination blob
    HttpWebRequest request = (HttpWebRequest)
    WebRequest.Create(destinationSASUrl);
    request.ContentLength = 0;
    request.Headers.Add("x-ms-version", "2014-02-14");
    /*add source information - this will be snapshot in
    our case. The uri query parameter of snapshot will
    be different for your snapshot */
    request.Headers.Add("x-ms-copy-source", sasUrl +
    "&snapshot=2014-09-19T05:29:50.4570000Z");
    request.Method = "PUT";

    using (HttpWebResponse respCreateSnapshot =
    (HttpWebResponse)request.GetResponse())
    {
      if (respCreateSnapshot.StatusCode ==
      HttpStatusCode.Accepted)//create operation
      returns ACCEPTED response
      {
        if (respCreateSnapshot.Headers != null)
        {
          //retrive copy state information from header
          copyStatus = respCreateSnapshot.Headers.
          Get("x-ms-copy-status");
```

```
            while (copyStatus != "success")
            {
              copyStatus = respCreateSnapshot.Headers.
              Get("x-ms-copy-status");
              Thread.Sleep(500);
            }
          }
        }
      }
      Console.WriteLine("restore operation status- " +
      copyStatus);
    }
    catch (Exception ex) { throw ex; }
}
```

So essentially restoring blob from snapshot is nothing more than a copy blob operation. In the above code sample, if I provide the destination blob name as original blob only (in above case it is "AzureBootCamp.zip") then original blob will be replaced entirely by the snapshot. This is how we restore the original blob. I don't want my original blob to be replaced therefore I performed copy operation with different destination blob name.

Note- *If you use the original blob to perform copy operation then all the associated snapshots of the original blob will not get copied.*

DELETE AZURE BLOB SNAPSHOT

The way we used the copy operation of blob storage to perform restore operation; similarly to delete blob snapshot, we need to use Delete Blob operation. Here too, I am retrieving the list of available snapshots to perform delete operation on it. First I am calling List blob snapshot operation to list total number of snapshots of the blob and after the delete operation, I again list the remaining snapshots. As expected, the deleted snapshot will not appear in the listing after the delete operation occurs. If your blob does not have any snapshot associated, then you may end up with *404 Not Found exception* or *Object reference not to set to instance of an object*.

We can delete individual snapshots for a blob. However, to delete a blob having associated snapshots, it is mandatory to delete all the associated snapshots first. If you try to delete the blob without deleting its snapshots then you will end up with *409 Conflict exception*. This is what is demonstrated in the code we will see shortly for the client library based snapshot delete operation. To specify delete operation on snapshot, enum `DeleteSnapshotsOption` can be used. Using this enum, you can specify options whether you wish to delete all snapshots but retain a blob OR delete blob along with all snapshots. Following is the demonstration of deleting of snapshots but retaining blob.

A. USING CLIENT LIBRARY

```
private static void DeleteSnapshotForBlob()
{
  try
  {
    //list the total number of snapshot present
    //currently
    ListSnapshotsForBlob();

    //specifying container name and blob name - change
    //them as per your blob and container name
    string containerName = "mycontainer";
    string blobName = "AzureBootCamp.zip";

    // Retrieve storage account from connection string
    CloudStorageAccount storageAccount =
    CloudStorageAccount.Parse(storageConnectionString);

    // Create the blob client
    CloudBlobClient blobClient = storageAccount.
    CreateCloudBlobClient();

    // Get a reference to the container and blob
    //snapshot
    CloudBlobContainer container = blobClient.
    GetContainerReference(containerName);
    DateTimeOffset? snapshotTime = ((CloudBlockBlob)
    (container.ListBlobs(null, true,
    BlobListingDetails.Snapshots).Where(x =>
    ((CloudBlockBlob)x).IsSnapshot).FirstOrDefault())).
    SnapshotTime;

    CloudBlockBlob snapshot = container.
    GetBlockBlobReference(blobName, snapshotTime);

    #region Delete individual snapshot
```

```
    //delete individual snapshot
    snapshot.Delete();
    Console.WriteLine("Individual Snapshot delete
    operation complete");

    //list total number snapshot remaining after
    //deleting one
    ListSnapshotsForBlob();

    #endregion

    #region Delete all snapshots but retain original
    blob

    CloudBlockBlob blob = container.
    GetBlockBlobReference(blobName);
    blob.Delete(DeleteSnapshotsOption.
    DeleteSnapshotsOnly);
    ListSnapshotsForBlob();

    #endregion
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

B. USING REST API

```
private static void DeleteSnapshotForBlobUsingREST()
{
  try
  {
    //specifying container name and blob name - change
    //them as per your blob and container name
    string containerName = "mycontainer";
    //my source and destination containers are same.
    string blobName = "AzureBootCamp.zip";

    //to perform any operation first lets generate the
    //SAS url on the source container-blob, validity 1
    //minute
    SASManager sasMgr = new
    SASManager(storageConnectionString);
    string sasUrl = sasMgr.
    GetAdHocWriteOperationSAS(containerName, blobName);

    //create request for destination blob
    HttpWebRequest request = (HttpWebRequest)
    WebRequest.Create(sasUrl);
    request.ContentLength = 0;
    request.Headers.Add("x-ms-version", "2014-02-14");
    //specify to delete all snapshots but retain the
    //blob
    request.Headers.Add("x-ms-delete-snapshots",
    "only");
    request.Method = "DELETE";

    using (HttpWebResponse respCreateSnapshot =
    (HttpWebResponse)request.GetResponse())
    {
      if (respCreateSnapshot.StatusCode ==
      HttpStatusCode.Accepted)//delete operation
      returns ACCEPTED response
      {
      }
    }
    Console.WriteLine("delete snapshots operation
    successfull");
    ListSnapshotsForBlob();
  }
  catch (Exception ex) { throw ex; }
}
```

COSTING MODEL OF BLOB SNAPSHOTS

Costing model of blob snapshots is very interesting. However before understanding costing model of snapshot, it is important to understand overall costing model of blob storage first. The blob storage costing is based on 3 main drivers –

1. Bandwidth or data transfer – This means data ingress and outgress. In other words you are charged for data going in the data center and data coming out of data center. Bandwidth has another important aspect here. Data ingress is FREE, data transfer in/out inside the same data center is FREE, however data coming out of the data center is charged.

2. Transaction – This means you are charged based on number of operations you perform against blob storage. For example, if you read or write to blob storage, you are charged for one transaction.

3. Capacity – This means the amount of storage consumed by your data. For example, if you say my blob is 5GB means you will be charged for 5GB of capacity.

Now with the understanding of these main factors of blob storage cost, let's understand the costing of Azure Blob snapshots.

1. Exactly identical

Bandwidth and transaction cost for snapshots is very similar to that of blob storage; however capacity cost is way different than the blob storage. As long as your snapshot "is exactly identical to original blob", no additional cost of capacity is incurred for snapshots. However the meaning of "exact identical to original blob" is very important. For capacity, you are always charged based on number of unique blocks for block type of blobs and number of unique pages for page type of blobs. If you have any snapshot same as original blob and if you update a single block from base blob, you will be charged for corresponding unmatched blocks of snapshot as well. To understand this scenario understand the following snapshot:

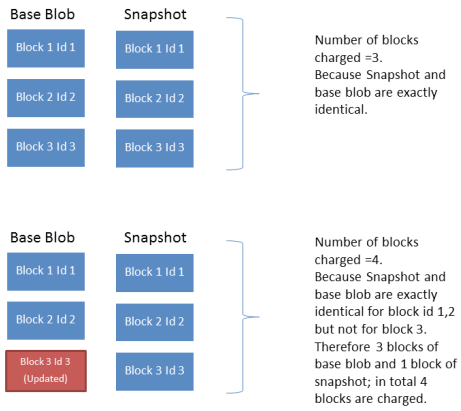


Figure 6: Costing Scenario

Azure blob storage does not have a way by which we can determine if the blocks are identical in all respects. Therefore whenever an update is performed for the block, irrespective of the same id or data, the updated blocks are treated as unique and hence charged subsequently.

2. Expensive methods for Snapshots

All methods having Upload keyword are expensive for snapshots. For example UploadFile, UploadText, UploadStream, UploadByteArray are the methods which replaces all the blocks in a blob, thereby making your snapshot and base blob exactly NON-IDENTICAL to each other. Hence you will be charged for base blob, plus all the snapshots.

BEST PRACTICE FOR SNAPSHOTS MANAGEMENT

When you want to perform an update to the blob, it is recommended to delete all the associated snapshots of the blob using client library or REST option as demonstrated above and then create snapshots again. This ensures base blob and snapshots are identical and no extra charge is incurred to you.

As far as possible, try to use PutBlock and PutBlockList method to perform incremental updates to the blob storage instead of a complete replacement of blob using upload specific methods.

CONCLUSION

Snapshots are the best way to provide read only access, avoid accidental writes and restore blobs to previous state. If used wisely, they can prove very beneficial in terms of performance ■

Download the entire source code from our GitHub Repository at bit.ly/dncm15-azuresnapshot

Kunal Chandratre is a Microsoft Azure MVP. He is working as Azure SME in leading software company in (Pune) India. He also works as a Freelancer with various organizations for Azure support and provides quick start trainings on Azure to corporates and individuals on weekends. He regularly blogs about his Azure experience and is a very active member in various Microsoft Communities and also participates as a 'Speaker' in many events. You can follow him on Twitter at: @kunalchandratre or subscribe to his blog - http://sanganakauthority.blogspot.com

GETTING STARTED WITH MUSTACHE.JS FOR TEMPLATING

One of the central tenets of modern web development is the separation of structure/content from behavior, a.k.a separate view from code. In our projects, we either tend to craft our HTML elements using a lot of string processing and concatenation, or build elements using native DOM methods and then insert these elements into the DOM. However as the size and complexity of a project grows, maintainability becomes an issue in such cases.

Templates can be beneficial here as templates simplify the entire process, increase reusability and minimize the amount of code needed to generate HTML elements from data.

Mustache.js (http://mustache.github.io/) is a logic-less templating library, which essentially supports the philosophy of little or no logic in your HTML templates. That means in your templates, there are no loops or if-else statements, instead there are only tags. There are several other logic-less templates out there like dust.js, handlebars.js etc which provide more functionality. We even have jQuery templating libraries like jQuery Templates and jsViews, which have been deprecated and superseded by the jsRender project. However we are choosing mustache.js for our article as I find it very simple to use and moreover we do not require too much complexity in this example. Having said that, feel free to look into the other templating engines too and decide what's best for your project.

The premise of mustache.js is quite simple. Think of it as a blueprint of HTML markup which lets you flow data on a page. You can use as many templates as you want on a page and take the same data and display it in different ways.

To get started, you can either download the library from Github https://github.com/janl/mustache.js or use a CDN from cdnjs.com.

In this example, we will read JSON data containing author information and display it on a page using Mustache template. Nothing fancy! The structure of the JSON data looks like this:

Mustache.js is a logic-less templating library, which essentially supports the philosophy of little or no logic in your HTML templates.

```
{ "authors" : [
{
    "name": "Suprotim Agarwal",
    "bio": "Suprotim is the founder of DotNetCurry.com, DNC .NET Magazine, SqlServerC
    "image": "suprotim.jpg",
    "twitter": "@suprotimagarwal"
},
{
    "name": "Alex Can",
    "bio": "Alex works as an author, freelance trainer and consultant on various clie
    "image": "alex.jpg",
    "twitter": "@twitter"
},
{
    "name": "Lynda Wasden",
    "bio": "Lynda has over 15 years of experience in IT education and development. Sh
    "image": "lynda.jpg",
    "twitter": "@twitter"
}
]
}
```

Figure 1: Sample JSON Template

Note: *Make sure your JSON is valid. Use a JSON validator like jsonlint.com to validate your JSON.*

There are various methods for defining mustache templates in your application, like using it inline or using it as external templates. For our example, we will create it inline but you should make it a habit of keeping it in a separate file for production applications, for better maintainability. So if you decide to keep it in a separate file, all you have to do is create a <templatename>.js file and then add a reference to this file using the `<script>` tag

```
<script src="../scripts/templatename.js"></script>
```

Coming back to our example, our template is called *authortmpl* and we will define template data inside of a <script> tag with a MIME type of text/template.

```
<script id="authortmpl" type="text/template">
</script>
```

Note: *We are using a MIME type other than text/javascript to prevent the browser from executing the template code.*

First off, we will start by creating a placeholder for authors. This is done using brace characters ({}) which looks like sideway mustaches (hence the name mustache.js).

```
<script id="authortmpl" type="text/template">
  {{#authors}}
  {{/authors}}
</script>
```

Observe how we are closing the *authors* placeholder using a backslash (/). This is similar to what we do with HTML elements. If you observe, authors is the name of the object in our JSON data file authorslist.json. Mustache.js cycles through all the authors in that object and applies the template that is defined here.

```
<script id="authortmpl" type="text/template">
    {{#authors}}
    <div class="auth">
        <h2>{{name}}</h2>
        <img src="images/{{image}}" alt="{{name}}"/>
        <p>{{bio}}</p>
        <p>{{twitter}}</p>
    </div>
    {{/authors}}
</script>
```

In the template, we have a div with a class **auth** and the author name, image, bio and website defined inside the div. If you observe, the tags defined inside the double braces match the elements in our .json file. Eg: {{name}} acts as placeholder which targets the **name** element inside the .json file and so on. Overall this template is nothing more than some html and placeholders. As you can see, it is a logic-less template.

Finally we need to use jQuery to grab our *authorslist* JSON file and insert the data into our template. We will use jQuery's $.getJSON() method to load JSON-encoded data from the server using a GET HTTP request.

```
$.getJSON('scripts/authorslist.json', function (data) {
  var templ = $('#authortmpl').html();
  var mustch = Mustache.to_html(templ, data);
  $("#authorlist").html(mustch);
});
```

Here the *getJSON* function loads the data file *authorlist.json* and executes a function literal to read the contents of the JSON file, into a *data* object. A variable called *templ* loads the content of the *authortmpl* template that we created a short while ago. We then use the Mustache.to_html() and pass the template and JSON data as parameters. Moustache will process the data and feed it into the template and create some HTML for us that we load in an *authorlist* div.

Go ahead and run the example. Here's the output:

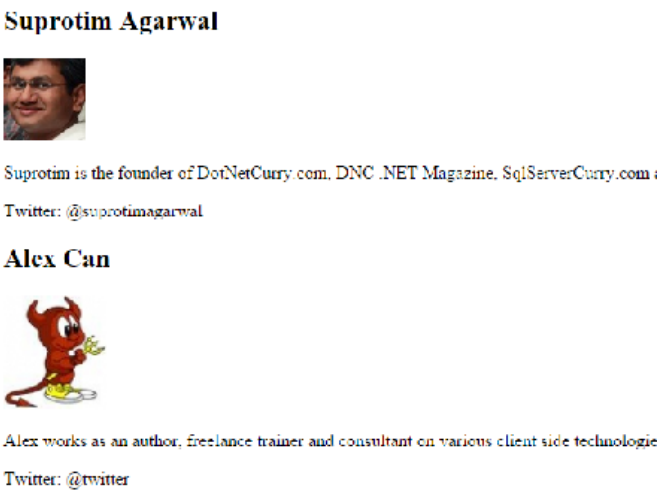


Figure 2: HTML Rendering using Mustache.

Feel free to add some CSS and beautify the example!

In this article, we saw that Moustache.js is a simple yet popular JavaScript template solution for modern web apps that consume and display data ■

This article was taken from my upcoming ***The Absolutely Awesome jQuery Cookbook*** at www.jquerycookbook.com

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