ISSUE 19 | JULY- AUG 2015

DNCMAGAZINE

www.dotnetcurry.com

33rd ANNIVERSARY

TABLE OF CONTENTS

TEAM AND CREDITS

Editor In Chief
Suprotim Agarwal
suprotimagarwal@a2zknowledgevisuals.com

Art Director
Minal Agarwal
minalagarwal@a2zknowledgevisuals.com

Contributing Authors
Ahmed Ilyas
Craig Berntson
Filip W
Kent Boogaart
Kunal Chandratre
Mahesh Sabnis
Punit Ganshani
Ravi Kiran
Shoban Kumar
Subodh Sohoni

Technical Reviewers
Damir Arh
Mahesh Sabnis
Shoban Kumar
Suprotim Agarwal

Next Edition
1st Sep 2015
www.dotnetcurry.com/magazine

FROM THE EDITOR

Suprotim Agarwal
Editor in Chief

If you want to grow old running websites or publishing magazines, ***you can't do it alone***. Today when we reach this milestone of 3 years, my note isn't meant to be a retrospective nor am I looking out to share any future plans. I simply want to express my gratitude, if you will.

At DotNetCurry (DNC), we have a 'simple goal' - to help you to learn, prepare and stay ahead of the curve. We aim to offer different prespectives, detailed walkthroughs and peeks into Microsoft and JavaScript technologies that you can use, directly and indirectly and on a daily basis, in your job.

Although as simple as it might sound, for us, it means digesting the latest developments in these exciting technologies as they metamorphize, and asking ourselves the same tough question about every piece of information, technology or feature that we come across: "Does this make sense to our readers?"

Fortunately my team comprises of Microsoft MVPs and invaluable industry experts. Together, we endeavour this goal. From the bottom of my heart, I want to thank all my fellow authors and reviewers for their time, energy and dedication.

Of course, the most important thank-you of all goes to you, the reader. After all, without YOU, this magazine would not exist. On behalf of the entire DotNetCurry (DNC) Magazine team, **Thank You** for reading us.

Please do continue sending us your invaluable feedback. Reach out to us on twitter with our handle @dotnetcurry or email me at **suprotimagarwal@dotnetcurry.com**

A **special note of thanks** to the following experts who shared their knowledge with the Developer community in the last year.

Craig Berntson
Mahesh Sabnis
Ravi Kiran
Subodh Sohoni
Gouri Sohoni
Minal Agarwal
Gil Fink
Raj Aththanayake
Kunal Chandratre
Vikram Pendse
Todd Crenshaw
Shoban Kumar
Rion Williams
Punit Ganshani
Pravinkumar Dabade
Nish Anil
Kent Boogaart
Irvin Dominin
Filip W
Filip Ekberg
Edin Kapic
Damir Arh
Ahmed Ilyas

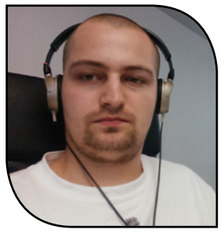
THANK YOU

FOR THE THIRD ANNIVERSARY EDITION


@sravi_kiran


@maheshdotnet


@craigber


@filip_woj


@ganshani


@subodhsohoni


@suprotimagarwal


@shobankr


@kent_boogaart


@kunalchandratre


@damirarh


@ahmedailyas


@saffronstroke

WRITE FOR US

ASP.NET MVC CONTROLS

WORK EFFORTLESSLY WITH ASP.NET MVC

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

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

DOWNLOAD FREE TRIAL

INFRAGISTICS

TRANSITIONING FROM

ASP.NET WEB API 2 TO MVC 6

This article will look at the parallels between the Web API framework pipeline and the upcoming MVC 6 framework - attempting to ease your transition into the new ASP.NET 5 world.

While some of the concepts (controllers, filters and so on) may look similar at first glance, because of Microsoft's approach to rewrite everything from scratch, transitioning your applications from current Web API application model to MVC 6 is actually not that simple.

There are obviously plenty of online - and offline - resources from both Microsoft and the community, discussing various new features of the ASP.NET MVC 6 framework. However, if you are worried about backward compatibility issues, porting your custom pipeline modifications or understanding that gap between Web API 2 and MVC 6 - then this article is for you.

WebApiCompatibilityShim

The first place to look at when you need to migrate an existing Web API 2 project to MVC 6 is the package called `Microsoft.AspNet.Mvc.WebApiCompatShim`.

ASP.NET MVC 6 is a very extensible framework - and `WebApiCompatShim` provides just that - a nice bridge between the old Web API 2 world and the modern reality of ASP.NET 5.

It is a layer of framework settings and conventions that transform the default MVC 6 behaviour into behaviour you might be more used to, from Web API 2.

What's interesting is that even the namespace of the types that `WebApiCompatShim` introduces, that are missing in MVC 6 but existed in Web API; is the same as it used to be in the Web API framework. Additionally, the shim also references System.Net.Http for you in order to reintroduce the familar classes of `HttpRequestMessage` and `HttpResponseMessage`.

All of this to minimize the pain related to porting your application. To enable the shim, you call the following extension method at your application startup, inside the `ConfigureServices` methods:

```
services.AddWebApiConventions();
```

ApiController

MVC 6 does not have dedicated controllers for Web API - this is because it unifies both MVC (web pages) and Web API (web apis).

`WebApiCompatShim` contains an `ApiController` class which mimics the structure of the Web API `ApiController`. It contains similar properties that your old Web API code might be using, such as User, Request; as well as same action helpers - methods returning IHttpActionResult.

While for new projects it is advisable to avoid `ApiController` and simply build around the common MVC 6 `Controller` base class (or even leverage the POCO-controller capabilities), if you are migrating from Web API 2, and a lot of your code was leaning on the members exposed by `ApiController` in Web API, it certainly can save you lots of hassle.

An interesting note is that `ApiController` in MVC 6 is actually a POCO-controller itself - it does not inherit from `Controller` base class.

HttpRequestMessage

`WebApiCompatShim` introduces a customized binder, `HttpRequestMessageModelBinder`, which allows you to bind an instance of `System.Net.Http.HttpRequestMessage` directly as a parameter of your action. This behavior was possible in Web API and you might have some existing code relying on this capability.

```
[Route("cars")]
public async Task<Ienumerable<Car>>
Get(HttpRequestMessage request)
{
  //access "request" parameter
}
```

You don't need to do anything special to enable this behavior - as long as `WebApiCompatShim` is configured, you can simply use `HttpRequestMessage` in any action. Additionally, an instance of `HttpRequestMessage` is also exposed as a `Request` property of the `ApiController`.

This is really useful, as there are a lot of tools i.e. logging ones, that work directly with `HttpRequestMessage` instance, rather than the new `Microsoft.AspNet.Http.HttpContext` around which ASP.NET 5 is built.

The aforementioned `HttpRequestMessageModelBinder` will take care of creating an instance of `HttpRequestMessage` from the `HttpContext` on every request.

On top of all that, the `WebApiCompatShim` will also introduce several extension methods that people commonly used in ASP.NET Web API. They are listed here:

```
public static class
HttpRequestMessageExtensions
{
  public static HttpResponseMessage
  CreateErrorResponse(this
  HttpRequestMessage request,
  InvalidByteRangeException
  invalidByteRangeException);

  public static HttpResponseMessage
  CreateErrorResponse(this
  HttpRequestMessage request,
  HttpStatusCode statusCode, string
  message);

  public static HttpResponseMessage
  CreateErrorResponse(this
  HttpRequestMessage request,
  HttpStatusCode statusCode, string
  message, Exception exception);

  public static HttpResponseMessage
  CreateErrorResponse(this
  HttpRequestMessage request,
  HttpStatusCode statusCode, Exception
  exception);

  public static HttpResponseMessage
  CreateErrorResponse(this
  HttpRequestMessage request,
  HttpStatusCode statusCode,
```

```
ModelStateDictionary modelState);

public static HttpResponseMessage
CreateErrorResponse(this
HttpRequestMessage request,
HttpStatusCode statusCode, HttpError
error);

public static HttpResponseMessage
CreateResponse<T>(this
HttpRequestMessage request, T value);

public static HttpResponseMessage
CreateResponse<T>(this
HttpRequestMessage request,
HttpStatusCode statusCode, T value);

public static HttpResponseMessage
CreateResponse<T>(this
HttpRequestMessage request,
HttpStatusCode statusCode, T value,
IEnumerable<MediaTypeFormatter>
formatters);
public static HttpResponseMessage
CreateResponse<T>(this
HttpRequestMessage request,
HttpStatusCode statusCode, T value,
string mediaType);

public static HttpResponseMessage
CreateResponse<T>(this
HttpRequestMessage request,
HttpStatusCode statusCode, T value,
MediaTypeHeaderValue mediaType);

public static HttpResponseMessage
CreateResponse<T>(this
HttpRequestMessage request,
HttpStatusCode statusCode, T value,
MediaTypeFormatter formatter);

public static HttpResponseMessage
CreateResponse<T>(this
HttpRequestMessage request,
HttpStatusCode statusCode, T value,
MediaTypeFormatter formatter, string
mediaType);

public static HttpResponseMessage
CreateResponse<T>(this
HttpRequestMessage request,
HttpStatusCode statusCode, T
value, MediaTypeFormatter formatter,
MediaTypeHeaderValue mediaType);
}
```

As you can see, the extension methods include a bunch of types that were Web API specific (`MediaTypeFormatter`, `HttpError`) or System.Net.Http specific (`MediaTypeHeaderValue`).

The shim introduces several other pain-relieving mechanisms, all of which I have discussed extensively in an article at http://www.strathweb.com/2015/01/migrating-asp-net-web-api-mvc-6-exploring-web-api-compatibility-shim/.

Old concepts mapped to new world

Message Handlers vs Middleware

In ASP.NET Web API, the way of dealing with cross-cutting concerns, such as logging or caching was through message handlers. The concept of message handlers was very simple, but very powerful. The handlers were chained one after the other, and each got a chance to process the incoming HTTP request and outgoing HTTP response. Together they formed a so called "Russian doll model", because the handler that interacted with the request first (the outer-most handler), got to interact with the response last.

The example here shows a typical Web API message handler (created off the base `DelegatingHandler` class).

```
public class MyMessageHandler :
DelegatingHandler
{
  protected async override
  Task<HttpResponseMessage> SendAsync(
  HttpRequestMessage request,
  CancellationToken cancellationToken)
  {
    //process request

    // Call the next (inner) handler.
    var response = await base.
    SendAsync(request,
    cancellationToken);

    //process response
    return response;
  }
}
```

The idea of handlers is sadly no longer present in ASP.NET 5, and the closest concept that can allow you to achieve same behavior and support the same functionality are OWIN middleware components. While going into the details of OWIN middleware is beyond the scope of this article, the fact that ASP.NET 5 is a de facto OWIN implementation allows you to plug in middleware in front of the MVC pipeline.

This is also a key difference between middleware and message handlers - middleware components are technically speaking, located outside of MVC 6 pipeline, while message handlers were located inside the Web API pipeline (albeit, at its very forefront, before controller selection).

There are various ways of writing middleware - a raw middleware has a rather unfriendly structure shown below:

```
Func<Func<IDictionary<string, object>,
Task>, Func<IDictionary<string, object>,
Task>>
```

You can, however, write ASP.NET 5 middleware in an object-oriented way, using the strongly typed `HttpContext` from the new HTTP abstractions. It will be the same `HttpContext` with which you get to interact i.e. inside of MVC 6 controllers. This is shown below.

```
public class MyMiddleware
{
  private readonly RequestDelegate _
  next;

  public MyMiddleware(RequestDelegate
  next)
  {
    _next = next;
  }

  public async Task Invoke(HttpContext
  context)
  {
    //process context.Request

    await _next(context);

    //process context.Response
  }
}
```

Web API Filters vs MVC 6 Filters

Filters were the typical component used in Web API to wrap the functionality of your actions with some extra logic. Web API 2 had 4 types of filters:

- action filters - for general aspect oriented functionalities, allowing you to invoke code before and after action execution
- authentication filters - introduced in Web API 2, allowed you to leverage host-level authentication provided via OWIN security middleware
- authorization filters - used for authorization (and in Web API 1, quite often, also for (!) authentication)
- exception filters - exception handling on an action-level

In ASP.NET 5, all the authentication logic has been moved to the *Microsoft.AspNet.Authentication* and related *Microsoft.AspNet.Authentication.** packages. This is where all the security middleware - direct successors to Katana Project security components are located. ASP.NET MVC 6 integrates with them deeply, and as a result, there is no need for authentication filters anymore.

ASP.NET MVC 6 still uses action filters (`IActionFilter`), exception filters (`IExceptionFilter`) and auhtorization filters (`IAuthorizationFilter`), and they are virtually identical to those in Web API 2, so the transition in those areas should be quite straight forward.

Interestingly, aside from the "classic" filters mentioned above, ASP.NET MVC 6 introduces two extra types of filters, which are really specialized versions of `IActionFilters`:

- TypeFilters
- ServiceFilter

`TypeFilters` allow you to have a custom filter class instantiated on demand for each request (potentially with extra parameters). Consider the following example:

```
[TypeFilter(typeof(MyFilter))]public
ActionResult Process();
```

This allows you to write a `MyFilter` class, implementing `IFilter` itself, and have it instantiated for you on every request. Why would you want this? To have constructor dependency injection into your filter! Normally this was not allowed in Web API without heavily modifying the filter pipeline. In this case, `MyFilter` could look as follows:

```
public class MyFilter : IActionFilter
{
 private IMyFancyService _fancy;

 public MyFilter(IMyFancyService fancy)
 {
  _fancy = fancy;
 }

 public void
 OnActionExecuted(ActionExecutedContext
 context)
 {
  //do stuff
 }

 public void
 OnActionExecuting(ActionExecutingContext
 context)
 {
  //do stuff
 }
}
```

In other words, `IMyFancyService` above is actually being resolved from the built-in ASP.NET 5 IoC container. You can also pass some extra arguments to the constructor of `MyFilter` explicitly - `TypeFilter` exposes `Arguments` object array for that.

ServiceFilters are similar, only a bit simpler. Consider the following signature:

```
[ServiceFilter(typeof(OtherFilter))]
public ActionResult Process();
```

In this case, `OtherFilter` will be immediately resolved from the ASP.NET 5 IoC container (where you'd normally register services) - hence the name "service". Obviously, that type would also need to implement `IFilter`. Because the filter in this case would be attempted to be obtained from the container, it must have been registered in it in the first place, otherwise this code would produce an error. On the contrary, in our previous example, our `MyFilter` class didn't need to be registered in the container - however it could consume dependencies that were coming from the container.

One extra feature worth mentioning is that MVC 6 filters are sortable out-of-the-box, so each filter will have an `Order` property. Web API 2 lacked this feature and that was actually quite constraining.

DependencyResolver vs built in dependency injection

In ASP.NET Web API, the most common way to enable dependency injection was to use the `IDependencyResolver` abstraction. Out-of-the-box, DI was not possible - you had to explicitly enable it by plugging in one of the community-provided implementations; pretty much every major IoC container for .NET had its own `IDependencyResolver` version.

Another option in Web API was to create a custom `IHttpControllerActivator`, which was responsible for instantiating controllers. There you could either resolve them by hand, or from a global DI container of your choice.

As far as dependency injection into other components in the Web API pipeline - message handlers, filters, formatters - constructor dependency injection was generally not supported. The only reasonable workaround was to use a service-locator like approach - obtain an instance of a registered `IDependencyResolver` from the current `HttpRequestMessage` and use it to resolve the necessary dependencies.

In ASP.NET MVC 6 things are much cleaner - as the underlying ASP.NET 5 runtime supports dependency injection out-of-the-box, without the need of plugging in an external IoC container. The built-in ASP.NET 5 IoC container is not very sophisticated, but it will fill the majority of needs most of the time.

You configure the ASP.NET 5 DI in the `Startup` class, inside the `ConfigureServices` method.

```
public void
ConfigureServices(IServiceCollection
services)
{
  services.AddMvc();
  services.AddSingleton<IService>(new
  MyService());
  services.AddTransient<IAnotherService,
  AnotherService>();
}
```

With the set up shown above, you can inject both `IService` and `IAnotherService` into any controller through a constructor (inlcuding POCO controllers!). You can also inject the dependencies into a controller's property, using the [`FromServices`] attribute.

```
public class MyController : Controller
{
  [FromServices]
  public IService Service { get; set; }

  //rest of controller omitted for
  brevity
}
```

These dependencies will also be hydrated if you inject them into the `Invoke` method of middleware components:

```
public async Task Invoke(HttpContext
context, IService service)
{
  //do stuff with service

  await _next(context);
}
```

Finally, if you create `ServiceFilterAttributes` or `TypeFilterAttributes`, you can also inject your dependencies into them through constructor.

ASP.NET MVC 6 also supports external IoC adapters - and those can be used to govern the dependency injection resolution in your application instead of the built in one. In that case, you'd use a different version of `ConfigureServices` method. The example here shows a simple set up of Autofac container:

```
public IServiceProvider
ConfigureServices(IServiceCollection
services)
{
  //create Autofac container build
  var builder = new ContainerBuilder();

  //populate the container with services
  here..
  builder.RegisterType<IService>().
  As<MyService>().
  InstancePerLifetimeScope();

  //build container
  var container = builder.Build();

  //return service provider
  return container.
  Resolve<IServiceProvider>();
}
```

Different model binding

The out-of-the-box model binding behavior in MVC 6 is much different than that of Web API and is more closely aligned with the MVC 5 model.

MVC 5 used to model bind over *everything* in the request - that is both query string and body of the request. In other words, a complex type that is being accepted as a parameter of your action could have one property bound from the body and another from the query string. On the other hand, Web API provided strict differentiation between the body - which was unbuffered and read using formatters, and query parameters which were read using model binders.

MVC 6 also uses binding and formatters, however the behavior is much different from what you might be used to when building APIs with Web API. Consider the following model and a controller:

```
public class Item
{
  public int Quantity { get; set; }
  public string Name { get; set; }
}

public class ItemController
{
  [Route("item")]
  public void Post(Item item) {}
}
```

If your action accepts this model as an input parameter, the following requests are valid in MVC6:

- `POST /item?quantity=1&name=foo, empty body`
- `POST /item?quantity=1, body: name=foo, content-type: application/x-www-form-urlencoded`
- `POST /item, body: name=foo&quantity=1, content-type: application/x-www-form-urlencoded`

In other words, you are able to mix-and-match between body parameters and query parameters, however - and this is very important - **this approach will only work for application/x-www-form-urlencoded content type**. If you try to send your body as JSON or XML, the default MVC 6 input binding behavior will not pick it up, and only bind from query string! MVC 6 does not even support working with JSON out of the box - the following request will not be bound correctly:

```
POST /item
Content-Type: application/json
{"name": "foo", "quantity": 1}
```

This is obviously much different from the traditional Web API behavior and can come back to haunt you if you are not aware of these model binding changes.

In order to force your API to use formatters to deserialize JSON/XML input from the body, you have to explicitly decorate your action parameter with `[FromBody]` attribute.

```
[Route("item")]
public void Post([FromBody]Item item) {}
```

You can also do it globally, for your entire application, which changes the default behavior of MVC 6 to more Web API-like. The code below utilizes an extensibility point called `IApplicationModelConvention` that allows you to iterate through all controllers, actions and parameters of all MVC 6 endpoints at the application startup and programmatically alter them. Because of that, you can inject `FromBodyAttribute` without having to manually add it everywhere. In there you could also establish your own convention or logic about when this given attribute should be applied.

```
public class
FromBodyApplicationModelConvention :
IApplicationModelConvention {
 public void Apply(ApplicationModel
 application)
 {
  foreach (var controller in
  application.Controllers)
  {
   foreach (var action in controller.
   Actions)
   {
    foreach (var parameter in action.
    Parameters)
    {
     if (parameter.BinderMetadata is
     IBinderMetadata ||
     ValueProviderResult.
     CanConvertFromString(parameter.
     ParameterInfo.ParameterType))
     {
      // behavior configured or simple
      type so do nothing
     }
     else
     {
      // Complex types are by-default
      from the body.
      parameter.BinderMetadata = new
      FromBodyAttribute();
     }
    }
   }
  }
 }
}
```

Then you have to enable this convention in the `Startup` class.

```
services.Configure<MvcOptions>(opt =>
{
  opt.ApplicationModelConventions.
  Add(new
  FromBodyApplicationModelConvention());
});
```

Similarily as in Web API, once you have decided to use `[FromBody]` in your action, you can only read one thing from the body, that is the following signature will be invalid:

```
[Route("test")]
public void Post([FromBody]Item item1,
[FromBody]Item item2) {}
```

On the other hand, if you remove `[FromBody]` from one of the parameters, you can still utilize query string and body binding together:

```
[Route("test")]
public void Post([FromBody]Item item1,
Item item2) {} //item1 from body, item2
from querystring
POST /item?name=nameforitem2&quantity-4
Content-Type: application/json
{"name": "nameforitem1", "quantity": 1}
```

Formatters in MVC 6

Since MVC 6 is a unified MVC/Web API framework, it is very natural that the concept of formatters (previously only present in Web API) was brought forward to the new world.

However, in MVC 6, formatters have been split into input- and output-formatters, instead of the traditional single, bi-directional formatter types that Web API used. On top of that, I have already mentioned in the previous section that in order to force MVC 6 to use input formatters in the first place, you must annotate your action parameter with `[FromBody]`.

Aside from that, the formatters are pretty much the same - they have to declare two things:

- a flag whether they can read / write a given type - since a specific formatter may not be applicable to a specific type, for example not every user type can be represented as RSS
- they read from request stream / write to response stream process

Output formatters should also define which content types they support. The formatter interfaces are listed here:

```
public interface IOutputFormatter
{
  IReadOnlyList<MediaTypeHeaderValue>
  GetSupportedContentTypes(
    Type declaredType,
    Type runtimeType,
    MediaTypeHeaderValue contentType);
  bool
CanWriteResult(OutputFormatterContext
context, MediaTypeHeaderValue
contentType);
  Task WriteAsync(OutputFormatterContext
context);
}
public interface IInputFormatter
{
  bool CanRead(InputFormatterContext
context);
  Task<object>
ReadAsync(InputFormatterContext
context);
}
```

What is important to remember, as opposed to Web API, is that XML formatters are disabled by default in MVC 6. If your API needs to support XML, you will need to bring in the Microsoft.AspNet.Mvc.Xml package and add XmlFormatters by hand:

```
services.Configure<MvcOptions>(options =>
{
 //this adds both inout and output
       formatters

 options.
 AddXmlDataContractSerializerFormatter();
});
```

Instead, out of the box MVC 6 contains only one input formatter and three output formatters:

- `JsonInputFormatter` / `JsonOutputFormatter`
- `PlainTex tFormatter` - which didn't exist in Web API and kicks in when you return a `string` from your actions, producing `text/plain` response. This is an important distinction, as Web API would produce an `application/json` response
- `HttpNoContentOutputFormatter` - also did not exist in Web API. It is responsible for creating a response with status code 204 when your action returns a `void` or `Task`. The behavior in this case is identical to Web API, however it was handled differently there (through a service called `ResultConverter`)

IHttpActionResult vs IActionResult

One of the popular additions to ASP.NET Web

API 2 was `IHttpActionResult`, which was used as a return type from an action, and was really an interface representing various factories for `HttpResponseMessage`. In fact, Web API shipped with a large number of implementations of `IHttpActionResult` in the box that you could use straight away in your API - such as `BadRequestResult` or `CreatedNegotiatedContentResult<T>` to name a few.

On the other hand, classic MVC framework has long had `ActionResult` as a base abstract type representing various responses. MVC 6 has been aligned closely with that approach, and `IActionResult` is the new abstraction that should be used in your actions. Since Web API and MVC frameworks have been unified in MVC 6, various `IActionResult` implementations can handle both traditional API scenarios (i.e. content negotiated object responses, No Content responses) and traditional MVC scenarios (i.e. view results).

```
public interface IActionResult
{
  Task ExecuteResultAsync(ActionContext
context);
}
```

Obviously, there are plenty of implementations of that interface that come bundled in the framework itself, many of which can be mapped one to one to `IHttpActionResult` implementations from Web API - for example `BadRequestObjectResult` is the rough equivalent of the old `BadRequestResult`. When building HTTP APIs with MVC 6, you will find yourself working the most with `ObjectResult` which is the type of `IActionResult` that has content negotiation built in - more on that later.

Interestingly, when working with Web API, one of the annoyances was the lack of I`HttpActionResults` that dealt with serving files, and more generally speaking, binary content. Such components would have to be built by hand or referenced through open source packages that grew around Web API in its vibrant community. This has been addressed in MVC 6, which introduces several specialized action results in that area - `FileContentResult`, `FilePathResult`, `FileResult` and `FileStreamResult.`

Content negotiation

In ASP.NET Web API, content negotiation was handled by `IContentNegotiator` interface. Normally, you would not use it directly, as the framework would handle content negotiation for you - when you returned a POCO object from an action (instead of `HttpResponseMessage` or `IHttpActionResult`), or when you called one of the extension methods on the `HttpRequestMessage` (i.e. `CreateResponse`). However, in some scenarios (for example determining the response media types for caching purposes) you'd need direct access to the ConNeg engine.

To facilitate these types of (now legacy) use cases, `WebApiCompatibilityShim` actually reintroduces all of the types involved in the content negotiation - so not only `IContentNegotiator`, but also `DefaultContentNegotiator` (the default implementation) or `ContentNegotiationResult`. So all your old content negotiation code can be pretty much ported verbatim to ASP.NET MVC 6.

On the other hand, ASP.NET MVC 6 does not have explicit service dedicated to running content negotiation. Instead, the logic responsible for selecting the relevant formatter is baked into the `ObjectResult` class. Inside its `ExecuteResultAsync`, responsible for writing to the response stream, the framework will walk through the available formatters and select a relevant one.

The logic for choosing a formatter is similar to that in ASP.NET Web API, and based on the following order of precedence:

- `Accept` header
- `Content-Type` header
- selection based on type match

ASP.NET Web API also had a very useful concept of `MediaTypeMappings`. Mappings allowed you to override content negotiation for specific request structure - an extension (i.e. `.json`), querystring (i.e. `?format-json`) or a predefined header (i.e. `Format: json`).

While `MediaTypeMapping` as a base class (that's how it was used in Web API) has no direct counterpart in ASP.NET MVC 6, the notion of media type mapping is indeed present there. `MvcOptions` exposes an object called `FormatterMappings`, which you can use (through `SetMediaTypeMappingForFormat` method) to map specific media types to a predefined string such as `json` or `.json`.

```
options.FormatterMappings.
SetMediaTypeMappingForFormat(
  "pdf",
  MediaTypeHeaderValue.Parse("text/
  pdf"));
```

The whole mechanism is a little less extensible than it used to be in Web API - in order to use this, you are forced to create a route with a `format` placeholder or use a `format` querystring. Example routes are shown below:

```
app.UseMvc(routes =>
{
  routes.MapRoute("formatroute",
  "{controller}/{action}/{id}.
   {format?}", new { controller =
  "Home", action = "Index" });
  routes.MapRoute("optionalroute",
  "{controller}/{action}.{format?}",
  new { controller = "Home", action =
  "Index" });
});
```

What's interesting is that ASP.NET MVC 6 maps `json` format to `application/json` out of the box. This means that by simply appending a `?format=json` querystring to a request, you will always get a JSON response. You can disable this behavior by calling `ClearMediaTypeMappingForFormat` on `FormatterMappings` object.

```
options.FormatterMappings.
ClearMediaTypeMappingForFormat("json");
```

It is also worth mentioning, that if you wish MVC 6 to issue 406 (Not Acceptable) response codes for situations when content negotiation is unnsuccessful - the framework defaults to JSON responses if it can't determine the media type - you do it by simply inserting a new formatter, `HttpNotAcceptableOutputFormatter` to the formatters collection. This is slightly different approach from Web API, where this type of functionality was controlled by tweaking the `DefaultContentNegotiator` and passing `true` into its constructor (representing `excludeMatchOnTypeOnly` parameter).

```
services.Configure<MvcOptions>(options =>
{
  options.OutputFormatters.Insert(0, new
  HttpNotAcceptableOutputFormatter());
});
```

HttpConfiguration vs MvcOptions

Finally, we should also briefly touch on the changes in the configuration of the framework. In Web API, everything was controlled through the `HttpConfiguration` type, which was the gateway to all important Web API components:

- all services
- filters
- formatters
- properties dictionary
- routes (through extension methods)

In MVC 6 the configuration happens through `MvcOptions` object, which over the course of this article, we have grown familiar with already. The outline is shown below:

```
public class MvcOptions
{
  public AntiForgeryOptions
  AntiForgeryOptions {get; set;}
  public FormatterMappings
  FormatterMappings { get; }
  public ICollection<IFilter> Filters {
  get; private set; }

  public
  IList<OutputFormatterDescriptor>
  OutputFormatters { get; }
  public IList<InputFormatterDescriptor>
  InputFormatters { get; }

  public
IList<ExcludeValidationDescriptor>
  ValidationExcludeFilters { get; }

  public int MaxModelValidationErrors
  {get; set;}
```

```
  public IList<ModelBinderDescriptor>
  ModelBinders { get; }

  public
  IList<ModelValidatorProviderDescriptor>
  ModelValidatorProviders { get; }

  public IList<ViewEngineDescriptor>
  ViewEngines { get; }

  public
  IList<ValueProviderFactoryDescriptor>
  ValueProviderFactories { get; }

  public
  IList<IApplicationModelConvention>
  Conventions { get; }

  public bool RespectBrowserAcceptHeader
  { get; set; }

  public IDictionary<string, CacheProfile>
  CacheProfiles { get; }
}
```

In many ways, `MvcOptions` plays the same role as `HttpConfiguration`, providing a central configuration point for many runtime settings such as formatters, filters or model binders. Additionally, caching is also controlled through `MvcOptions` (something that was completely missing in Web API!).

As far as the actual services go (for example, `DefaultFilterProvider`, responsible for orchestrating the filter pipeline or `DefaultControllerActivator`, responsible for instantiating the controller types), they are registered against the ASP.NET 5 IoC container directly. When you call `AddMvc()` method on the `IServiceCollection` at the application startup (necessary to be able to use MVC 6 in your application at all), the framework will add all of the necessary runtime services to the container. You can then tweak or replace those by interacting with the container itself.

Conclusion

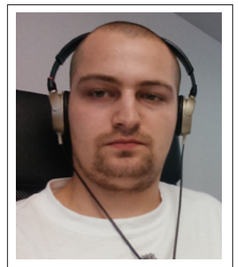
There are quite a few things to watch out for when trying to transition from HTTP API development using ASP.NET Web API to the new world of ASP.NET 5 and MVC 6. Aside from the obvious large scale changes, there are certainly some hidden landmines and subtle differences, and those are usually the most frustrating to deal with. However, if you pay attention to the points highlighted in this article, it shouldn't be too frustrating of a task.

Let me close out by sharing an interesting thought from Darrel Miller, who, himself an HTTP guru, has been one of the most active members of the ASP.NET Web API community. He said recently on Twitter:

> **"How many WebAPI people would be more likely to use MVC6's WebAPICompatShim if it was named WebAPIConventions?"**

And I think this hits the nail on the head. The Web API Compatibility Shim is a great way of enforcing specific coding standards and conventions, which have already proved to be useful and battle tested in ASP.NET Web API. Do not be wary about using them – even if you get shivers about a notion of using a "shim" in production. At the end of the day, the existence of the shim itself is just a testament to the extensibility of MVC 6 ■

About the Author



Filip W is a Microsoft MVP, popular ASP.NET blogger, open source contributor and an author of "ASP.NET Web API 2: Recipes". Specializes in ASP.NET Web Stack and modern Microsoft web technologies. Experienced in delivering robust web solutions in corporate context, worked on projects in many corners of the world (Canada, Switzerland, Finland, Poland, Scotland). Currently works for a Canadian digital agency, Climax Media. Follow him on Twitter @filip_woj.

CLIMAX

New Debugging Features in Visual Studio 2015

Debugging, diagnostics and profiling are fundamental stages in software development that ensures that all errors, bugs and performance bottlenecks are found and resolved, before the application is deployed. Every application developer is responsible for writing logical, syntactical and error free code. However in real life, errors and bugs creep in, and can slow down the application. For this reason, developers are always looking out for tools to manage code for debugging and performance.

Although some 3rd party profiling and debugging tools like RedGate's Ants Performance Profiler for .NET applications makes the job easy for you Visual Studio 2015 and recent Visual Studio 2013 updates have also introduced new capabilities to maximize developer productivity and improve code quality.

In this article, we will explore some of the enhancements made to debugging features in Visual Studio 2015 RC. We will also revisit some Visual Studio 2013 debugging features wherever applicable, for the sake of completeness.

Some new debugging features in Visual Studio 2015 RC are as follows:

- Understanding Debugging with Stepping
- Debugging a specific method out of multiple method calls from a single statement
- Debugging Code Data Visualizer
- Managing the Display of the Debugged Data
- Evaluating lambda expressions in debugger watch window

Since this article is based on the Release Candidate version of Visual Studio, it is possible that any feature discussed in this article may change in the future.

Before we get started with these features, let us set up a sample application.

Set up a Sample MVC Application for Debugging

Step 1: Open Visual Studio 2015 and create a new empty ASP.NET MVC application as shown in Figure 1:

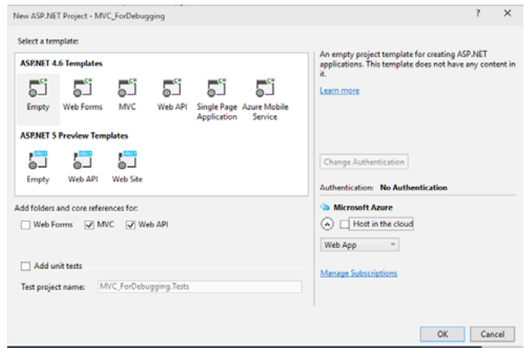


Figure 1: Empty ASP.NET MVC application

In this MVC application, add a new SQL Server Database of the name ApplicationDB.mdf in the App_Data folder. In this database, add a table called 'EmployeeInfo' with the following structure.

	Name	Data Type	Allow Nulls
🔑	EmpNo	int	☐
	EmpName	varchar(50)	☐
	Salary	int	☐
	DeptName	varchar(50)	☐
	Designation	varchar(50)	☐

Figure 2: EmployeeInfo table

Step 2: In the Models folder, add a new ADO.NET Entity Data Model with the name ApplicationDBEDMX.edmx. This step will start a wizard. Select ApplicationDB.mdf file and the EmpoyeeInfo table. After completing the steps of the wizard, the EmployeeInfo table mapping will be generated.

Step 3: In the same project, add a new folder with the name 'DataAccess'. In this project, add a new class file with the following code in it:

```
using System.Collections.Generic;
using System.Linq;
using MVC_ForDebugging.Models;
namespace MVC_ForDebugging.DataAccess
{
 public class EmployeeDAL
 {
  ApplicationDBEntities ctx;
  public EmployeeDAL()
  {
   ctx = new ApplicationDBEntities();
```

```
  }

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

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

  public void
  AddNewEmployee(EmployeeInfo emp)
  {
   ctx.EmployeeInfoes.Add(emp);
   ctx.SaveChanges();
  }

  public void UpdateEmployee(int id,
  EmployeeInfo emp) {
   var e = ctx.EmployeeInfoes.
   Find(id);
   if (e != null)
   {
    e.EmpName = emp.EmpName;
    e.Salary = emp.Salary;
    e.DeptName = emp.DeptName;
    e.Designation = emp.Designation;
    ctx.SaveChanges();
   }
  }

  public bool CheckEmpNameExist(string
  ename) {
   bool isExist = false;

   foreach (var item in ctx.
   EmployeeInfoes.ToList())
   {
    if (item.EmpName.Trim() ==
    ename.ToUpper().Trim())
    {
     isExist = true;
     break;
    }
   }
   return isExist;
  }

  public bool
  CheckValidSalForDesignation(int sal) {
   bool isValid = true;
   if (sal < 5000)
   {
    isValid = false;
   }
   return isValid;
  }

  public void DeleteEmployee(int id,
  EmployeeInfo emp) {
   var e = ctx.EmployeeInfoes.Find(id);
   if (e != null)
   {
    ctx.EmployeeInfoes.Remove(e);
    ctx.SaveChanges();
   }
  }
 }
}
```

This class will be used for performing CRUD operations using the Entity Framework model we added in Step 2.

Step 4: In the Controllers folder, add a new MVC controller with the name 'EmployeeInfoController'. Add the following code in it:

```
using System.Web.Mvc;
using MVC_ForDebugging.Models;
using MVC_ForDebugging.DataAccess;

namespace MVC_ForDebugging.Controllers {
public class EmployeeInfoController :
Controller {
 EmployeeDAL obj;
 public EmployeeInfoController()
 {
  obj = new EmployeeDAL();
 }

 // GET: EmployeeInfo
 public ActionResult Index()
 {
  var emps = obj.GetEmployees();
  return View(emps);
 }

 // GET: EmployeeInfo/Details/5
 public ActionResult Details(int id)
 {
  var emp = obj.GetEmployee(id);
  return View(emp);
 }
// GET: EmployeeInfo/Create
 public ActionResult Create()
 {
  var emp = new EmployeeInfo();
  return View(emp);
 }

 // POST: EmployeeInfo/Create
 [HttpPost]
public ActionResult
Create(EmployeeInfo emp)
{
 try
 {
  if (obj.CheckEmpNameExist(emp.
  EmpName) && obj.CheckValidSal
  (Emp.Salary))
  {
   obj.AddNewEmployee(emp);
   return RedirectToAction("Index");
  }
  else
  {
   return View(emp);
  }
 }
 catch
 {
  return View(emp);
 }
}

// GET: EmployeeInfo/Edit/5
public ActionResult Edit(int id)
{
 var emp = obj.GetEmployee(id);
 return View(emp);
}

// POST: EmployeeInfo/Edit/5
[HttpPost]
public ActionResult Edit(int id,
EmployeeInfo emp)
{
 try
 {
  obj.UpdateEmployee(id, emp);
  return RedirectToAction("Index");
 }
 catch
 {
  return View();
 }
}

// GET: EmployeeInfo/Delete/5
public ActionResult Delete(int id){
 var emp = obj.GetEmployee(id);
 return View(emp);
}

// POST: EmployeeInfo/Delete/5
[HttpPost]
public ActionResult Delete(int id,
EmployeeInfo emp){
 try {
  obj.DeleteEmployee(id, emp);
  return RedirectToAction("Index");
 }
 catch {
  return View(emp);
 }
 }
}
```

This controller class contains action methods which calls the methods from **EmployeeDAL** class added in Step 3.

Generate Views from Index, Create, Details and Edit action methods. To generate View, right click on the Action Method and select the **Add View** option from it. Build and run the application.

Note: Open RouteConfig.cs file from the App_Start folder and change the routing as shown in the following code:

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "EmployeeInfo", action = "Index",
            id = UrlParameter.Optional }
    );
}
```

Figure 3: Changing Routing information

This will display the Index View when the application runs.

Now that we have setup the application, let's jump into the new Debugging features in Visual Studio 2015.

New Debugging Features in Visual Studio 2015 RC

Understanding Debugging with Stepping

We will explore a new experience of stepping through the code. Open EmployeeInfoController.cs and apply breakpoint on the Index action method. Run the application, and you will see that the Index() method from the EmployeeInfoController

will be hit. We will make use of F10 (Step Over) and F11 (Step Into). We will use F10 to debug the GetEmployees() method call from Index() method and check values returned from it.

In Visual Studio 2015, we can make use of *Locals* and *Autos* window to check return values from the *GetEmployees()* method without jumping into it, as shown in Figure 4:

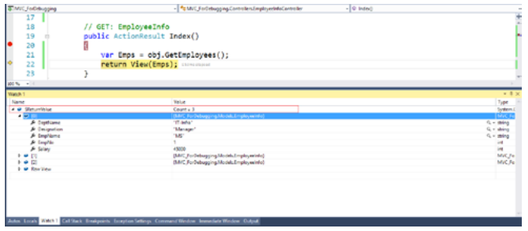


Figure 4: Checking return value in Locals Window

Using this feature, we can check the return values from an external method call without Stepping into the method. This is more useful if we are debugging in a multi-layered application where the code getting debugged, makes calls to multiple external methods.

Note: *Visual Studio 2015 IDE provides a handy performance tooltip feature called PerfView, using which we can check how much time is taken by a statement. Typically we need this in case of foreach loops. In Figure 4, you can see the PerfView tooltip (yellow marked) which shows that the statement took 1341ms to execute.*

Alternatively we can also make use of the Watch window to check returned values using pseudo variable of the name '$ReturnValue'. Open the Watch Window and use the $ReturnValue as shown in Figure 5.

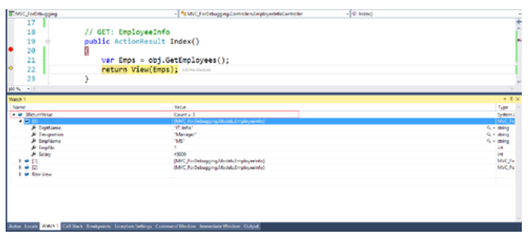


Figure 5: Checking return value using Watch window

Debugging a specific method from a statement with out of Multiple Method Calls

Consider a scenario in the debugging code, where we have a statement calling more than one method, as shown in Figure 6:

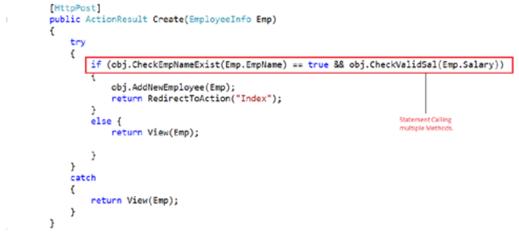


Figure 6: Statement calling more than one method

The *if* statement makes calls to *CheckEmpNameExist()* and *CheckValidSal()* methods (along with *get()* method call for EmpName and Salary properties). In traditional debugging, to debug these methods we need to put a breakpoint on every method. In case of Visual Studio 2015, we can choose a specific method to debug using **Step Into Specific** context menu.

To experience this new feature, apply a breakpoint on the *if* statement of the *Create()* action method of the EmployeeInfoController as shown in Figure 6. Run the Index View. Click on the *Create New* link of the Index View. This will bring up the Create view. Enter Employee data in Create view and click on the *Create* button. This will display the code in the debugger on the *if* statement as shown in Figure 7:

```
if (obj.CheckEmpNameExist(Emp.EmpName) == true && obj.CheckValidSal(Emp.Salary))
{
    obj.AddNewEmployee(Emp);
    return RedirectToAction("Index");
}
else {
    return View(Emp);

}
```

Figure 7: Debugger Code

Right-click on the debug statement and select **Step Into Specific** option as shown in Figure 8:

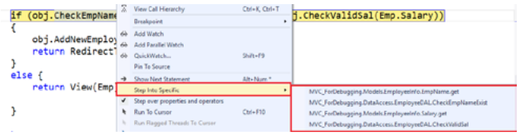


Figure 8: Step Into Specific option

Figure 8 shows a list of all methods called in the currently debugged statement. From here, we can now select specific methods to be debugged. This will apply one-time breakpoint to the entry-point to that method, and the method will be debugged.

Select CheckEmpNameExist() method from the list and this method will get debugged as shown here:

```
public bool CheckEmpNameExist(string ename)
{ 1ms elapsed
    bool IsExist = false;

    foreach (var item in ctx.EmployeeInfoes.ToList())
    {
        if (item.EmpName.Trim() == ename.ToUpper().Trim())
        {
            IsExist = true;
            break;
        }
    }
    return IsExist;
}
```

Figure 9: Debugging a method

This feature gives us the benefit of selection based debugging for a specific method. One important thing to note here is that we can directly jump to the specific statement for debugging using **Run to Cursor** option from the context menu (available in Visual Studio 2013 as well). In the *CheckEmpNameExist()* method, right-click the *if* statement and select Run to Cursor, and the if statement will be hit by the debugger. This eliminates the need to apply and remove breakpoint on a specific statement for debugging. To go back to the immediate statement from which we started the debugging, we can make use of the **Step Out** feature from the Debug menu.

Debugging Code Data Visualizer (available in VS 2013 too)

When debugging code, if the current statement contains a Data Source expression, we can view the data using Text, XML, HTML and JSON Visualizers.

To experience this feature, in the EmployeeInfoController.cs add the following namespace reference:

```
using System.Web.Script.Serialization;
```

Add the following lines in the Index Action method (highlighted)

```
public ActionResult Index()
{
  var Emps = obj.GetEmployees();

  var jObj = new JavaScriptSerializer();

  var jsonData =   jObj.Serialize(Emps);

  return View(Emps);
}
```

In the above method, add a breakpoint on the return statement. The code serializes *Emps* into JSON format. Run the application and place the mouse cursor on the **jsonData** variable. You will see a **magnifier** on debug as shown in Figure 10:

Figure 10: Debug magnifier

Click on the dropdown of the magnifier to display the data visualizer options window:

Figure 11: Data Visualizer Options

Select the JSON Visualizer to display a window with JSON data:

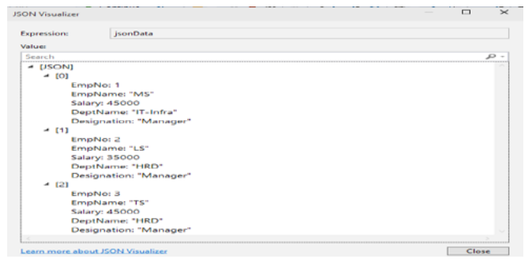


Figure 12: JSON Visualizer

Figure 13 below displays a *Search* window to search data from the received data.

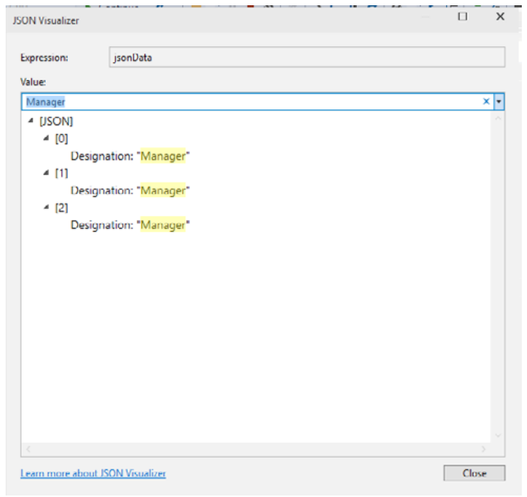


Figure 13: Search JSON Visualizer

The data visualizer window will remember the selection made by us e.g. if we select JSON visualizer, it will keep remembering the JSON visualizer option. Likewise we can take advantage of various data visualizers.

Managing the Display of the Debugged Data (also present in VS 2013)

While debugging the code using Visual Studio 2013 and Visual Studio 2015 RC, we are given a feature of managing the data display. In our code, the Index action method shows the Employees data. We can manage the display of this data during debugging using *DebuggerDisplay* attribute class. To experience this feature, open the EmployeeInfo class and apply the *DebuggerDisplay* attribute as shown here:

```
[DebuggerDisplay("Emp {EmpNo}")]
  public partial class EmployeeInfo
  {
    public int EmpNo { get; set; }
    public string EmpName { get; set; }
    public int Salary { get; set; }
    public string DeptName { get; set; }
    public string Designation { get;
    set; }
  }
```

In this code, the DebuggerDisplay accepts the string parameter. Here *Emp* represents the constant string and the {EmpNo} represents the EmpNo property which will represent the value of the Employee Record. Debug the Index action method and view the value for Emps as shown in Figure 14.

```
// GET: EmployeeInfo
public ActionResult Index()
{
    var Emps = obj.GetEmployees();
    var jOb  criptSerializer();
    var json j.Serialize(Emps);
    return V
}
```

Figure 14: Debugger display for Employee records

Figure 14 above shows the debugger display for Employee records as Emp 1, Emp 2, etc.

Using Make Object ID to Maintain the State of the Debug Result

Another useful debugging feature in VS 2015 (also present in VS 2013) is **Make Object ID**. While debugging the code, we make use of the *Watch* window to evaluate the expression. The watch window can maintain the state of the expression in the current scope only. When the expression moves out of the scope, the watch on it becomes invalid. But sometime it is necessary for us to keep watch on the data generated by the previous debug for the expression not in scope. We can implement this using the *Make Object Id* on the watch expression.

To experience this feature, apply a breakpoint on the Index and Create with HttpPost action methods. Run the application, the Index action method will be hit. Press F10 on the Emps = obj.GetEmployees(); expression. To add watch, right-click on *Emps* and select **Add Watch** from the context menu as shown in Figure 15:

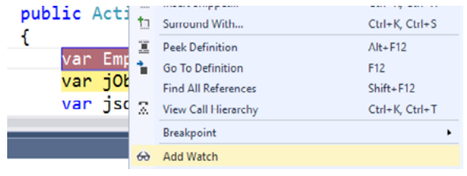


Figure 15: Add Watch option

This will display the watch window as shown in Figure 16.

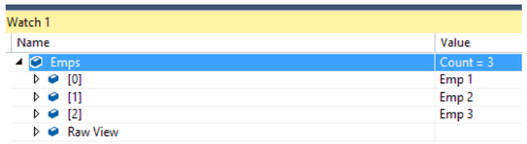


Figure 16: Add watch window

To maintain the state of a specific record e.g. *Emp 1,* right-click on the [0] record and select the **Make Object Id** option as shown in Figure 17.

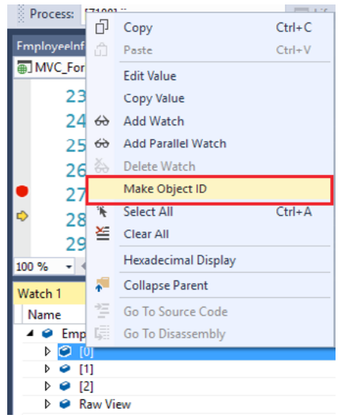
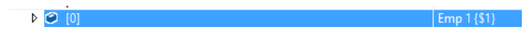


Figure 17: Generate ObjectID using Make Object ID

This will generate the Object Id,



..and will maintain the state of the first Employee record. Complete the debugging to display the Index view in the browser. In this view, click on the *Create New* link to display the Create view. This means now we have come out of the scope of the Index action method.

In the Create view, enter data in the TextBoxes and click on the Create button. This step will start debugging the Create action method. Once we enter in the debug mode, we can see the watch window where the previous watch is disabled as shown in Figure 18.

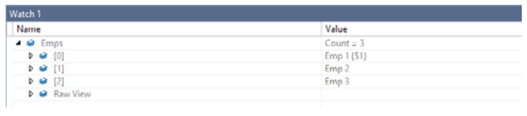


Figure 18: Disabled watch from previous debugging session

Here we cannot interact with Employee records Emp 2 and Emp 3. But since we have generated the object Id for Emp 1, we can check values generated using the Object Id as $1. In the watch Window, enter the *Name* as $1 and press enter, the values will be displayed as shown in Figure 19.

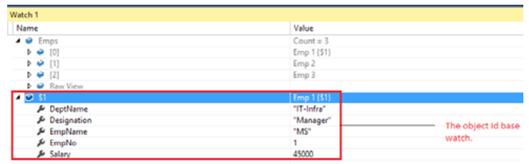


Figure 19: Object ID base watch

Evaluating lambda expressions in debugger watch window

While debugging code, sometimes we may come across LINQ queries or lambda expressions. Now we can evaluate these Lambda Expressions or LINQ queries using the *Immediate Window*. To experience this feature, we need to make some changes in the code.

Open EmployeeInfoController.cs and add the following reference in the code:

```
using System.Linq;
```

Apply the breakpoint on the Index() action method. Run the application and the Index method will be hit. Complete debugging for the following statement.

```
var emps = obj.GetEmployees();
```

Now open the **Immediate Window** and start typing code.

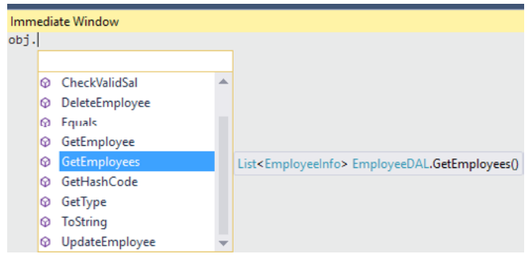


Figure 20: Lambda Expression IntelliSense

Yes! We have IntelliSense!!

The Lambda Expression with its result can be seen in Figure 21.

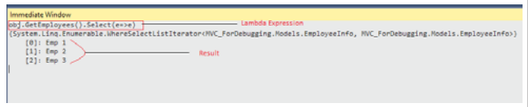


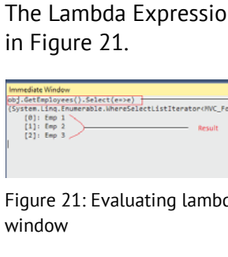
Figure 21: Evaluating lambda expressions in debugger watch window

And with this feature, we can now evaluate Lambda Expressions using the debugger **Immediate Window** in Visual Studio 2015.

Conclusion:

Visual Studio 2015 (currently in RC as of this writing) has provided some cool developer friendly and easy to use features for code debugging. Developers can now make use of these features for effective debugging management ■

Download the entire source code from GitHub at bit.ly/dncm19-vs2015debug



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

USING ADFS WITH AZURE FOR SINGLE SIGN-ON IN AN ASP.NET MVC APPLICATION

Introduction

Active Directory Federations Services (ADFS) is an enterprise-level identity and access management service provided by Microsoft. ADFS runs as a separate service and hence any application that supports WF-Federation and Security Assertion Markup Language (SAML), can leverage this federation authentication service.

In this article, we are going to use ADFS configured in Azure VM for Single Sign-on implementation. If you have never configured an ADFS in Azure VM or need to know the benefits of using an ADFS, along with some key terminologies, **I strongly recommend** you to read this article first. The steps shown in this article are also applicable to ADFS which already exists on-premises. For demonstration purposes, we are not using on-premises ADFS but are going to use Azure VM to act as our domain controller and ADFS server.

Applicable Technology Stack

1. Windows Server 2012 R2 DataCenter
2. Azure VM based ADFS OR on-premise ADFS
3. Self-Signed SSL Certificates
4. Visual Studio 2013 Community Edition

Important ADFS configurations

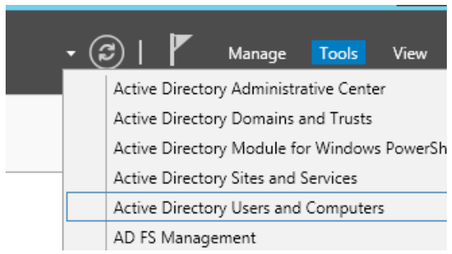
In this section, I will demonstrate what ADFS should consist of, in order to integrate well with your ASP.NET MVC application. I will be using the same setup as demonstrated over here.

Add details for existing AD user and create new user

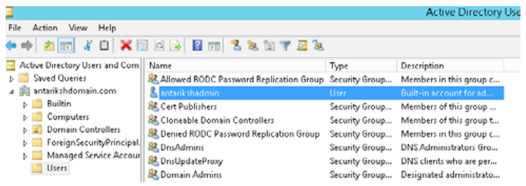
The MVC application we are developing is a claims-aware application, therefore ADFS should send in the claims that will represent information of the user on a successful authentication. To make this possible, important details of each ADFS user must be configured in Active Directory. This section highlights settings which are necessary for a user to enable him/her for use of claims-aware application.

Login to the Active Directory server. In my case, it is the Azure VM. In your case it may be Azure VM or on-premises AD server. Remember the steps are going to be same irrespective of Azure VM or on-premises Server. For this article, we will use Azure VM.

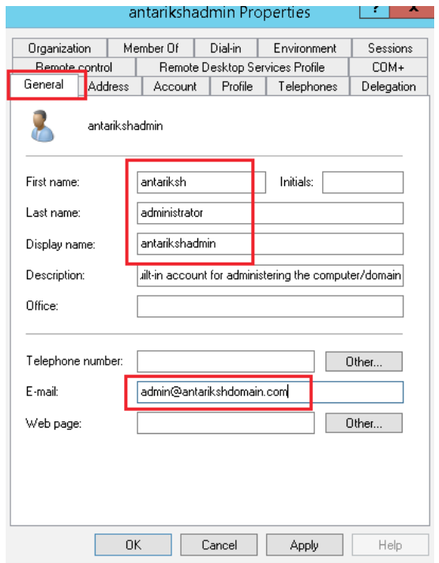
From "Server Manager" options > select "Tools" menu > and open "Active Directory Users and Computers" option.



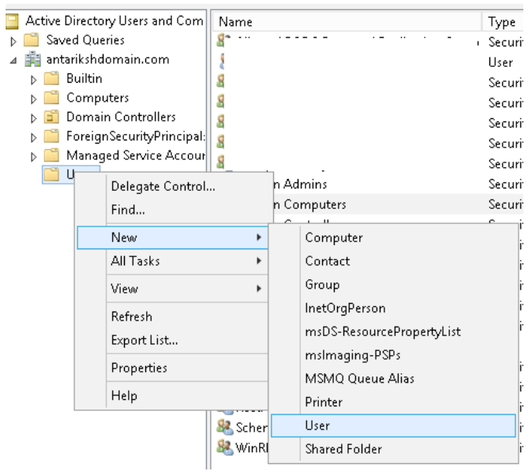
From "AntarikshDomain.com", locate the user "antarikshadmin".



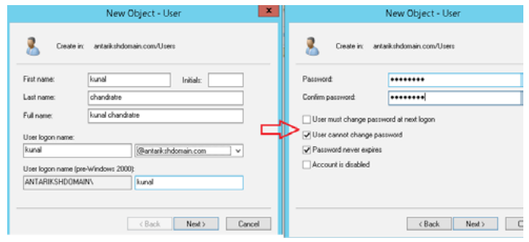
Double click on the user to open the details window. Fill in the details in the "General" tab as shown in the following screenshot.



Now we will add another user in the same active directory. Right click on "Users" in the left hand panel and select option as "New User".



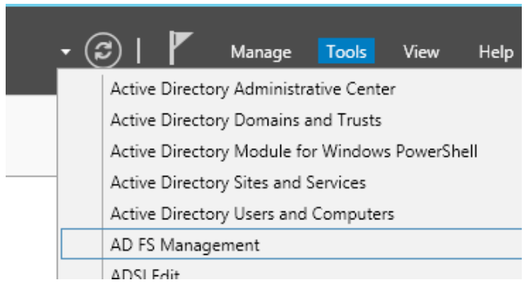
I am going to add my name 'kunal' as a new user. If you wish, you can add yours or anybody else's. Refer to the following screenshot for details. Click Next and provide a password of your choice. Make sure that you check the options "User cannot change password" and "Password Never Expires". Click Next to continue and then click on Finish to complete the new user creation procedure.



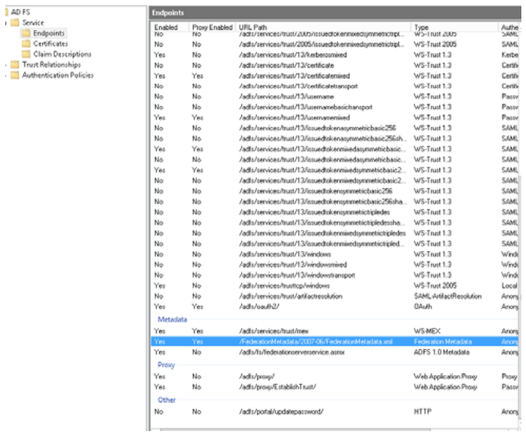
What is my Metadata URL and where to find it?

First let me show how you can locate the metadata URL of your ADFS and in the next section, we will explore the *why* part of it.

To get your metadata url, open Server Manager or Azure AD VM (or on-premises AD machine) > and from "Tools" option > select "ADFS Management" option as shown here –



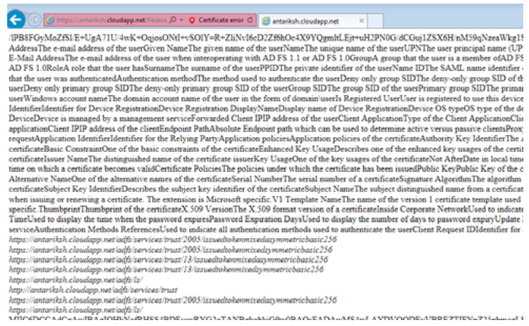
This step will bring up the "ADFS Management" window. Then expand the "Service" tab in the left hand panel and select "Endpoint" option. All endpoints available on ADFS will be displayed. Scroll to the bottom and you can locate the metadata URL as shown here –



If you observe, the metadata URL in the screenshot is being displayed as **/FederationMetadata/2007-06/FederationMetadata.xml.** This is only a part of the metadata URL. The full metadata URL would be your DNS name + the URL we just saw. As you know, in our case we are using Azure VM for hosting ADFS, therefore our Azure Cloud Service URL is nothing but our DNS name. Therefore in this case, the complete Metadata URL will be –

https://antariksh.cloudapp.net/FederationMetadata/2007-06/FederationMetadata.xml

If you have used the link I shared earlier to set up ADFS in Azure VM and followed all the steps till now, then the URL will work for you as it is. Now if you put this URL in a browser like IE, then this is what you will see –



It is absolutely fine if you are not able to make any sense out of this metadata. When we configure this metadata in our application, that's when we will realize its importance.

Note – If you face any certificate errors like "There is problem with this websites security certificate" then there's nothing to worry here. Just click on "Continue to this website (not recommended)." Option to continue. The reason this error gets displayed in the browser is because we are using Self-Signed Certificates.

If you have provided a different name to your cloud service, then your URL will be

https://YourCloudServiceName.cloudapp.net/FederationMetadata/2007-06/FederationMetadata.xml

If you are using on-premises ADFS, then you metadata URL will be –

https://YouDomainName.com/FederationMetadata/2007-06/FederationMetadata.xml

Before moving ahead, let's first understand why do we need this Metadata URL and what is its purpose. This is explained in the next section 'General Steps for Claim Based Authentication'.

General Steps for Claim Based Authentication

When we say that our application will use ADFS for authentication, it means that our application should be able to understand claims sent by ADFS. There are 4 generic steps involved –

Step 1 - Add logic to your application for supporting claims

When you are planning to build a claim-based application, your application needs to know how to validate security token coming from a security token service (in our case ADFS) and how to retrieve claim from that security token. Microsoft has provided rich classes in its Windows Identity Foundation (WIF) framework to handle implementation of claims-based application. The WIF framework provides a basic programming model for claim-based authentication. In order to use WIF, you will need to add a reference to the WIF assembly (Microsoft.IdentityModel.dll) in your WCF or ASP.NET application.

Step 2 - Acquire or build an issuer (ADFS)

I have already implemented this step over here Configuring ADFS in Azure Virtual Machine. Needless to mention, the same steps can be used if you are using on-premises ADFS.

ADFS will be issuing security tokens and claims, therefore ADFS is also termed as Issuer. In a broader view, all Identity Providers are treated as Issuer or Secure token Services (STS).

Step 3 – Configure application to trust the issuer (ADFS)

Once you have developed a claim-based application using WIF and have used ADFS to issue security token, the next step is to set up a trust relationship. This is a very important step. An application needs to trust the ADFS to identify and authenticate users and make claims about their identities. When you configure an application to rely on a specific issuer, you are establishing a TRUST (or TRUST RELATIONSHIP) with that issuer. The application needs to understand what claims ADFS will be offering, what key application should be using to validate the tokens? What URL the user should be using to request token? And so on.

ADFS provides answers to the above questions using "**federation metadata**". This is an XML document which ADFS provides to the application and in our case it can be obtained from the URL –

https://antariksh.cloudapp.net/FederationMetadata/2007-06/FederationMetadata.xml

I hope you now understand the important of the ADFS Metadata URL.

It includes a serialized copy of the issuer's certificate that provides your application with the correct public key to verify incoming tokens. WIF includes a wizard that automatically configures application's identity settings based on this metadata. We just need to give the wizard the URL for the ADFS, and it downloads the metadata and properly configures your application.

Step 4 – Configure the issuer (ADFS) to know about your application

The trust relationship is always two ways. In Step 3, we saw that by using Metadata URL, we can configure trust relationship between Applications and ADFS. Now ADFS needs to know a few things about an application before it can issue it any tokens. These questions include information related to URI of the application, which claims are required by applications out of issue by ADFS, URL of the application to receive tokens and so on.

ADFS can get this information about the application when we configure the application as Relying Party. Our application is going to rely on ADFS for security token and authentication therefore it is called as the **Relying Party**.

We will follow these 4 steps so as to implement ADFS integration in an ASP.NET MVC application.

Create ASP.NET MVC application secured by ADFS

So here we go with the actual implementation. I hope you have noted the technology stack I have mentioned at the beginning of the article.

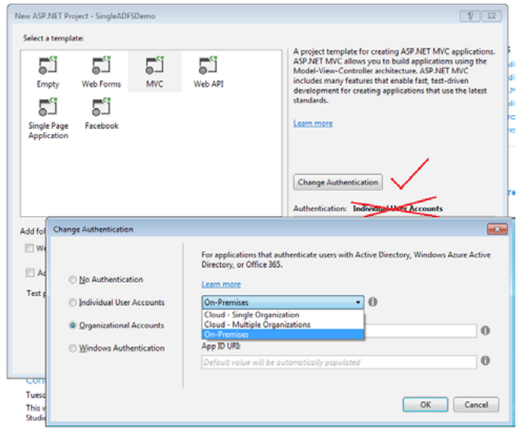
Create claims-aware ASP.NET MVC application

The first step would to be open Visual Studio 2013 in the administrator mode and click on File > New > Project. A dialog box appears where in you put the name of your project as **SingleADFSDemo** (or anything else of your choice) and specify the appropriate location. When we click OK, another pop up appears where we need to choose the template of the project. We will select MVC template here. As soon as we select the MVC template, automatically in the right hand panel, the authentication mode changes to "Individual User Account" and we don't want that!!

So click on "Change Authentication" button, another pop up appears where you can select the option as "Organization Account". In the right hand panel, a drop down appears with default value selected as "Cloud-Single Organization" and we don't want this option too!!

In the same drop down, select the option On-Premises and here is the catch/confusion! Let's understand these drop-down options one by one!

'



- **Cloud - Single Organization** – this option should be chosen if your application is authenticating against Azure Active Directory.

- **Cloud – Multiple Organization** – this option is for SaaS applications where one application will be used by multiple clients.

- **On – Premises** – This option allows you to connect to any WS-Federation provider (like ADFS) which offers Metadata document and this is our option for the article!

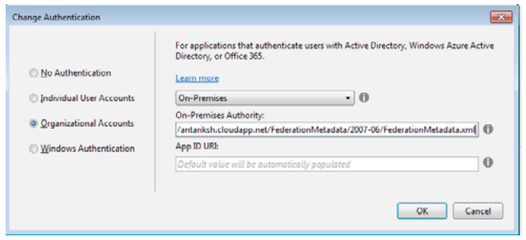
Note *– Don't go by the literal name On-Premises. This option suggests that you can use any corporate ADFS irrespective of the fact that ADFS is hosted on-premises or on Azure VM.*

So select "On-Premises" and the following screen appears asking for 2 types of information.

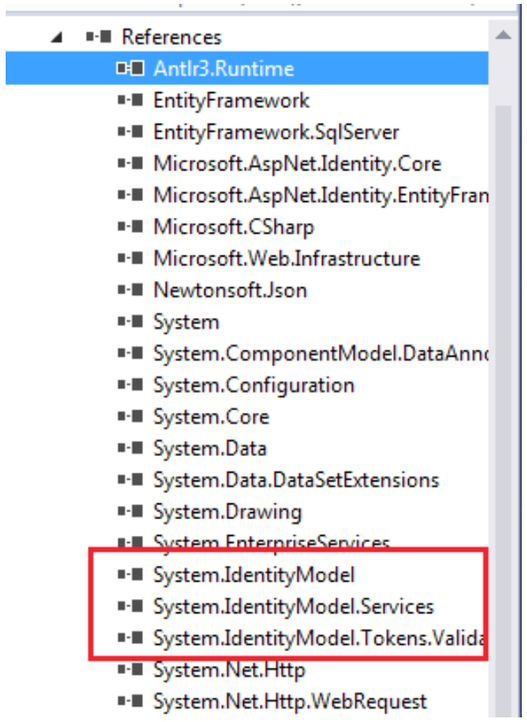
- **On Premises Authority** – this should be our ADFS metadata URL which can be from Azure VM based ADFS or On-premises hosted ADFS.

- **App Id URI** – This will be the URL assigned to the ASP.NET MVC project we are creating using IIS Express. If we host it on a custom domain like https://kunal.com then this URL of custom name will be my App Id URI. Of course this can be changed later after project creation using web.config. We are going to keep it blank and let it get auto-populated after project creation.

Add the metadata url (https://antariksh.cloudapp.net/FederationMetadata/2007-06/FederationMetadata.xml) of domain Antariksh, keep App Id URL blank and click on OK to continue. Keep rest of the options as it is and click on OK to create the project.



After project creation, expand the references from Solution explorer and you will observe that System.IdentityModel.dll reference has got added automatically.



In web.config an entry for `<system.identityModel>` gets added with all the settings required for making application claim-aware and trust the metadata of ADFS. It is also important to note here that localhost IIS express URL will be added under `<audienceUris>` tag and metadata URL, realm etc. added under `<appSettings>`.

The audience URL is the application URL so at a later date after finishing testing with localhost, let us say we wish to host the MVC application on:

- Azure cloud service, then my audience URI and realm would be https://mycloudservicename.cloudapp.net/ or

- Azure webApps, then it would be https://mywebsites.azurewebsites.net/ or

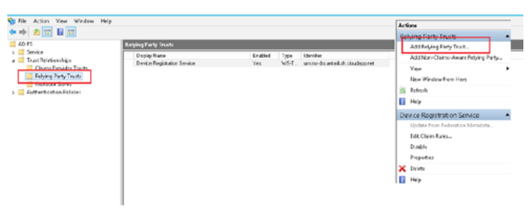
- Custom domain like kunal.com, then it would be https://kunal.com/ and so on. You got the idea right?

So hereby we have defined the trust relation in our MVC application or Relying Party to Azure VM based ADFS.

Adding MVC app as Relying party trust in ADFS

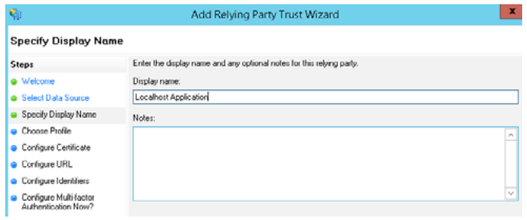
Let us move back to ADFS to do some configurations. In this step, we will essentially tell ADFS that our MVC application with localhost url is a trusted application and you can send the security tokens to it after successful authentication from a user. In technical terminology, this is nothing but **adding relying party trust in ADFS**.

So login to the ADFS Azure VM. Open Server Manager > Tools > ADFS Management. Expand "Trust Relationships" from left hand panel and select "Relying Party trusts" option. You will see that Device Registration Service is already present as a relying party. Now click on the option "Add Relying Party Trust" in the right hand panel.

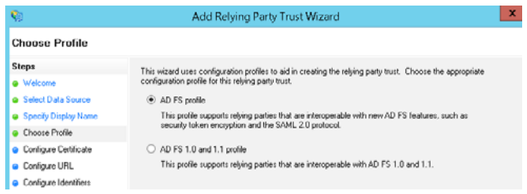


The Add relying party wizard will appear. Click on the Start button to continue. Select the option

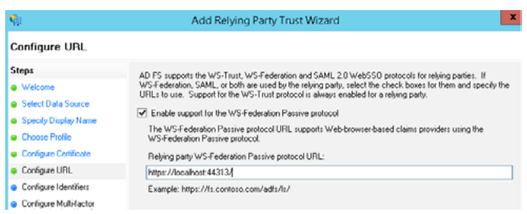
"Enter data about relying party manually" and click on Next to continue. Now we are going to use our localhost URL as relying party therefore we are providing the name as **localhost application**.



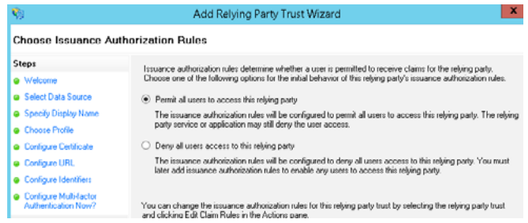
Click Next to continue. In the Choose Profile window, select "ADFS Profile" option and click Next to continue. The ADFS configured on Windows Server 2012 is ADFS 3.0 therefore we are not selecting 1.0 and 1.1 profile option.



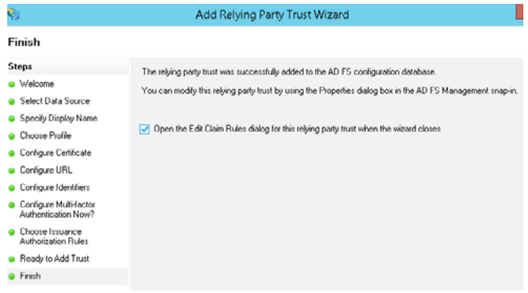
In the "Configure Certificate" option let's not do anything. This window gives you an option to choose the certificate for encrypting tokens. As of now, we are encrypting none of the tokens therefore simply click on Next to continue. In the "Configure URL" window, select the checkbox against the option "Enable support for the WS-Federation passive protocol". The Textbox will get enabled and this is where we need to put our relying party URL or in simple words- our MVC application URL. From web.config file, copy value for the key `ida:AudienceUri` and put in the textbox as shown here and then click Next to continue.



In "Configure Identifiers" we already have the required relying party added, therefore simply click Next to continue. Now select "I do not want to configure multi-factor authentication settings for this relying party trust at this time" and click Next to continue. Select "Permit all users to access this relying party" and click Next to continue.

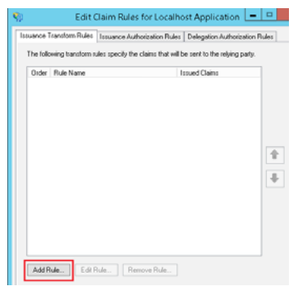


In the "Ready to add trust" window, click Next to continue. In the "Finish" window, select the checkbox to open the claims rules and click on Close.

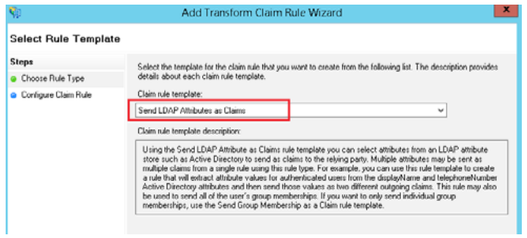


The Edit claim rules window pops up. At this point, ADFS knows about our MVC application but there a couple of additional things required. This is where we tell ADFS which claims need to be sent to the relying party and what values will be present in those claims.

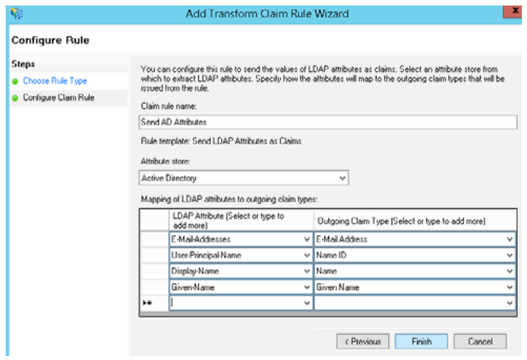
Click on the "Add Rule" button.



Select template value as "Send LDAP attributes as claims". Actually claims will be sent by Active Directory and Active Directory is a LDAP based store, therefore we are selecting this template. Now click on Next to continue.



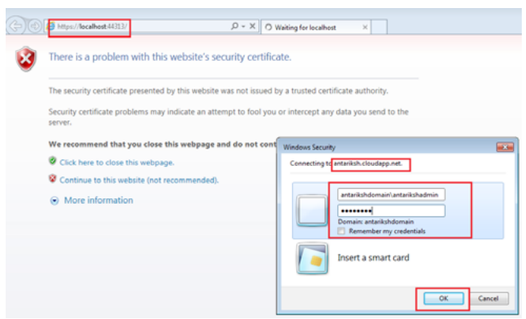
In configure Rule window, provide the name for the rule as **Send AD Attributes**. Select the attribute store as "Active Directory". In the mapping table, map the values as shown here –



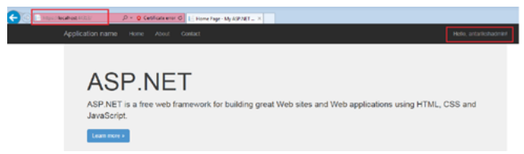
Click on Finish to exit the wizard and then click OK to complete claims rules configuration.

Running the application

Alright. Now is the time to test everything we have done so far. Going back to Visual Studio, hit F5 to start the application in debug mode. Click continue on certificate errors and hurraahh! If you are running your application in IE, you will see a credentials box from our Azure VM or on-premises ADFS as shown here –



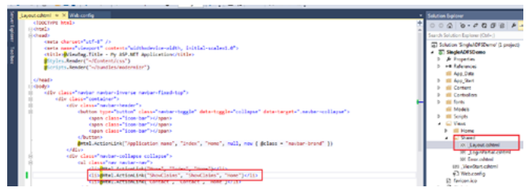
Enter the credentials of antarikshadmin or your ADFS admin and you will be redirected to the home page of the MVC application with your admin name flashing in the right hand upper corner as shown here –



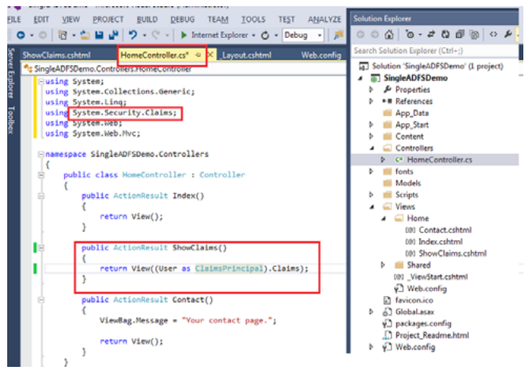
Isn't this amazing? We did a complete ADFS integration in our MVC application **without writing a single line of code!** Let's go further and analyze the claims we are receiving from ADFS.

Viewing the claims retrieved

Now we must understand the different claims our application is receiving. We are going to populate all the received claims in the About action. Let's first rename the About action to Showclaims from Layout.cshtml as highlighted here –

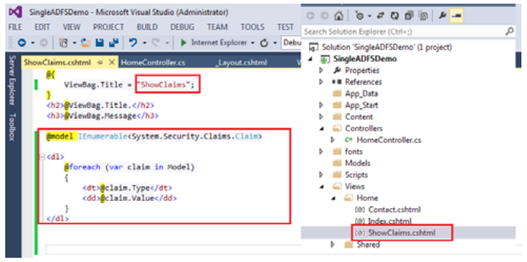


Modify the HomeController *About* action to ShowClaims and add some code to return claims to the views. You will need to use System.Security.Claims reference at the top. Refer to the following screenshot for details –

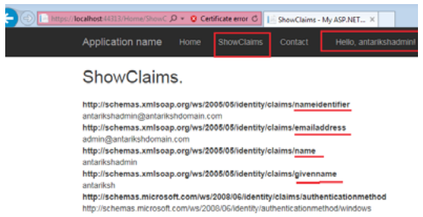


Rename the About.cshtml view to

ShowClaims.cshtml and add the following code in the view as shown here –



Run the application, enter ADFS admin credentials. After successful login, click ShowClaims action at the top and all the claims sent by ADFS will be displayed.



The way we used admin user, other AD users can also be used in the same way. Remember we configured an extra user in ADDC called 'kunal' or whatever you chose the username as. Go ahead and use the credentials of this new user created in ADDC and see if you get similar results.

Now you can use any of these claims like email in your application and can provide authorization to the user based on authorization roles and rules in your database.

Making life easier for developers to bypass ADFS authentication during development

One of the most frequent questions I have encountered in person and in forums is about developers asking how to bypass ADFS temporarily. Once ADFS integration is implemented in a project, every time you run the application, it keeps asking you to enter the credentials and Developers who are working everyday on the same project, are bound to get frustrated. That's where I thought it is worth to spend some time explaining a simple technique to bypass ADFS authentication mechanism during development work.

Just open the web.config, search for `<deny users="?" />` and comment it. You are done! Now if you run the application, ADFS login box will not appear and entire application will be accessed anonymously.

```
<system.web>
  <authentication mode="None" />
  <authorization>
    <!--<deny users="?" />-->
  </authorization>
  <compilation debug="true" targetFramework="4.5" />
  <httpRuntime targetFramework="4.5" requestValidationMode="4.5" />
</system.web>
```

Conclusion

In this article we saw how easy it was to implement a single ADFS integration in ASP.NET MVC and how to retrieve the claims, relying party, claims rules and many other aspects involved with a typical WSFederation implementation using Windows Identity Foundation (WIF) technique.

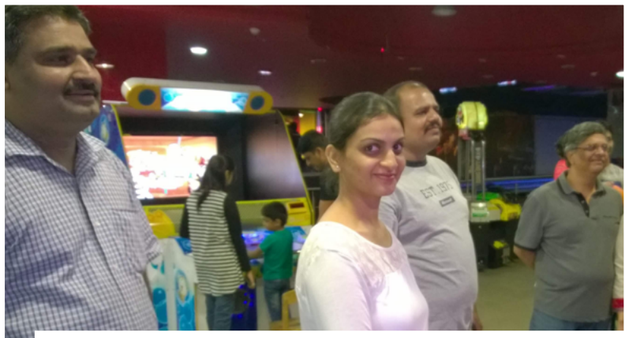
We can also configure our ASP.NET MVC application to consume **multiple** ADFS authentication (either Azure VM configured or On-premises) using Microsoft OWIN KATANA. Keep an eye out for an upcoming article on multiple ADFS authentication on www.dotnetcurry.com ■

About the Author

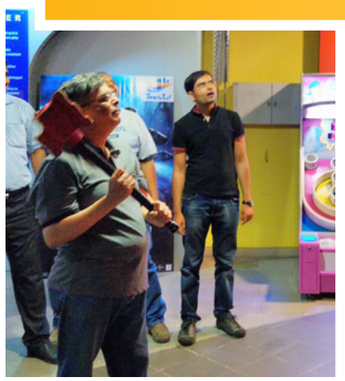
kunal chandratre

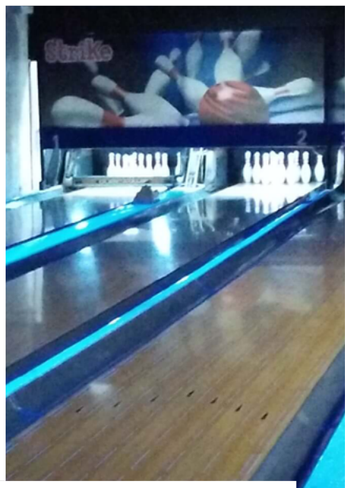
Microsoft® Most Valuable Professional

Kunal Chandratre is a Microsoft Azure MVP and works as an Azure Architect in a leading software company in (Pune) India. He is also an Azure Consultant to various organizations across the globe 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 at http://sanganakauthority.blogspot.com



WORKING HARD, PARTYING HARDER





WPF ItemsControl Fundamentals

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

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

In this first part of the article, we will explore the fundamental concepts of the `ItemsControl` class.

The Basics

The XAML for the simplest possible `ItemsControl` is:

```
<ItemsControl/>
```

This is equivalent to invoking the `ItemsControl` constructor and omitting any property modifications. If you put this inside a `Window`, what you get for your efforts is decidedly dull. Since the `ItemsControl` has no items (how could it? - we've not told it where to get items from) it renders without obvious appearance. It's still there, but we'll need to change the `Background` property to `Red` in order to reveal it:

```
<ItemsControl Background="Red"/>
```

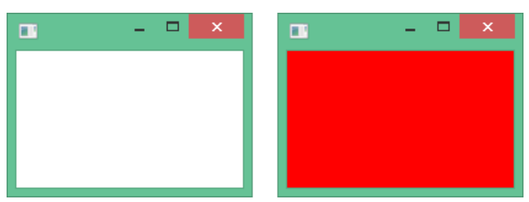


Figure 1 A default ItemsControl on the left, and with an explicit background color on the right

As you can see in Figure 1, it's now clear that our `ItemsControl` occupies the entire `Window`. This implies that it must be having some effect on the visual tree, and we can confirm this using Visual Studio 2015's new WPF Visualizer tool.

***Note**: Alternatively, you could use the excellent Snoop utility. See* http://snoopwpf.codeplex.com/.

In order to use this visualizer we need to be debugging, so we first need to set a breakpoint in the code-behind for our `Window`. But where? We only have a constructor at this point and during construction WPF has not yet had a chance to realize the visual tree, so we need to add some code. Perhaps the simplest thing to do is add this code to our constructor:

```
this.Loaded += delegate {
    var dummy = this;
};
```

Now we can set a breakpoint on our `dummy` variable. Execute the application and when the breakpoint is hit, hover your cursor over `this`. Click the little magnifying glass icon that appears in the tooltip. You will then see the WPF Visualizer, per Figure 2.

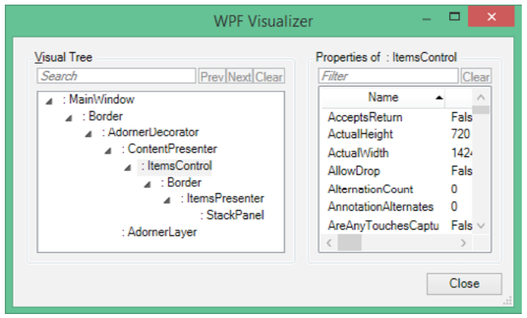


Figure 2 The WPF debug visualizer, new to Visual Studio 2015

As you can see, even a default `ItemsControl` with no items still includes some visual elements. The `Border` is what we see rendered when we set the `BackgroundColor` on our `ItemsControl`. Inside the `Border` resides an `ItemsPresenter` and, inside that, a `StackPanel`. Neither of these elements has any visual appearance themselves – they're only of utility if our `ItemsControl` actually has items.

Populating Items

The simplest way to get some items into our `ItemsControl` is via the `Items` property. This property is of type `ItemCollection`, which is essentially just a non-generic collection of items in the `ItemsControl`. As we'll discover later, different `ItemsControl` subclasses have different preferences for the type of items you place within them, but the `ItemsControl` itself doesn't care – as long as the item is a `FrameworkElement` subclass. For example:

```
<ItemsControl>
  <ItemsControl.Items>
    <Label>A Label</Label>
    <Button>A Button</Button>
    <CheckBox>A CheckBox</CheckBox>
  </ItemsControl.Items>
</ItemsControl>
```

We can simplify the XAML further because the `Items` property is the content property for an `ItemsControl`:

```
<ItemsControl>
  <Label>A Label</Label>
  <Button>A Button</Button>
  <CheckBox>A CheckBox</CheckBox>
</ItemsControl>
```

Either way, we get the UI depicted in Figure 3.

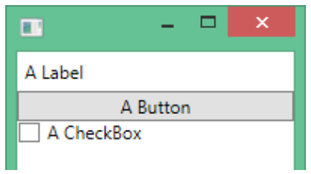


Figure 3 Some user interface items in an ItemsControl

What if we simply throw some textual content into an `ItemsControl` instead of UI components? Let's try throwing some places in there:

```
<ItemsControl>
  London
  Amsterdam
  Adelaide
</ItemsControl>
```

Figure 4 shows the result, which is perhaps a little unexpected.

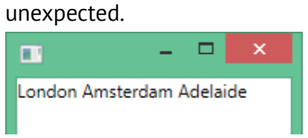


Figure 4 Some textual items in an ItemsControl

What happened here is that any text is automatically placed inside a `TextBlock`. Per the rules of XML parsing, all three "items" are parsed as one piece of text. We can confirm this via the WPF Visualizer – see Figure 5.

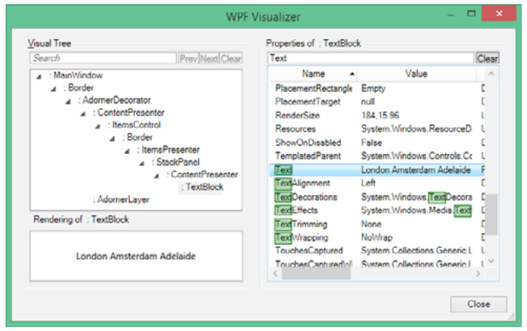


Figure 5 Our text has been hosted inside a single TextBlock

OK, so now we know we can add any number of user interface elements to an `ItemsControl` simply by including them as children of the `ItemsControl` element in our XAML. But what's the point? How is this any better than simply including the items as children of a `StackPanel` instead? In fact, if you look at Figure 5 you'll see that a `StackPanel` is hosting our items anyway (we'll find out why later).

The answer is: you normally wouldn't. At least, not with an `ItemsControl`. You might use this approach with subclasses of `ItemsControl`, for reasons we'll discover later. Regardless, it's an instructive stepping-stone on our path to a data-driven `ItemsControl`, which is where `ItemsSource` comes in.

The `Items` and `ItemsSource` properties are mutually exclusive – it makes sense to set only one of them and any attempt to use both will result in an exception. `ItemsSource` allows us to give the ItemsControl a data source from which to materialize the items it displays. This could be an XML document or a list of CLR objects. In practice I have found XML document data sources to be of use only in standalone demos, so I am going to ignore them here. In a production system, you will almost certainly want to create view models around your data – whether it's XML-based or otherwise – and bind your `ItemsControl` to them instead.

Let's start out by just assigning a `List<string>` to the `DataContext` of our `Window`:

```
public MainWindow()
{
  InitializeComponent();
  this.DataContext = new List<string>
  {
    "London",
    "Amsterdam",
    "Adelaide"
  };
}
```

Now we can modify our XAML thusly:

```
<ItemsControl ItemsSource="{Binding}"/>
```

The result is shown in Figure 6. It is visually identical to what we'd get if we manually added three `TextBlock` controls to the `Items` property of our `ItemsControl`. The resulting visual tree is also very similar, but not exactly the same. When using `ItemsSource`, each of our `TextBlock` controls is hosted inside a `ContentPresenter` whereas when using `Items` they are not. The reasons are not terribly important here, but it comes down to `ItemsControl` container generation logic, which is responsible for wrapping items in a container if required.

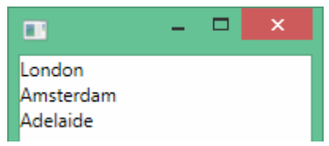


Figure 6 Our ItemsControl is now obtaining its data from a List<string>

`ItemsControl` also provides a `HasItems` property, but I haven't found it to be of any use. If you need to trigger UI changes based on your available data, you're better off modelling those requirements in your view models. Not only does this give you more centralized logic and greater flexibility (for example, what if you need to know when you have only one item?), it also enables you to test such scenarios too.

Now that we know how to get items into our `ItemsControl`, can we stop ignoring the fact that the items are visually boring? How can we adjust their appearance?

Basic Item Appearance Customization

A simple place to start with adjusting the appearance of our items is with the `ItemStringFormat` property. This property enables us to provide a format string that will be used to produce the displayed string for each of our items. For example, if we set it as follows:

```
<ItemsControl
  ItemsSource="{Binding}"
  ItemStringFormat="City: {0}"/>
```

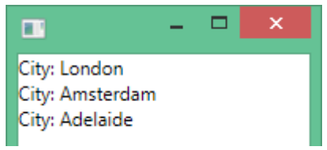


Figure 7 Using the ItemStringFormat property to modify the text shown for each item

The result is that each of our items is prefixed with "City: ", as you can see in Figure 7. Of course, all the usual rules and behavior for .NET string formatting apply here. In this case, our data items are of type `string`, so we're a little limited in our formatting capabilities. Let's change to using dates:

```
this.DataContext = new List<DateTime>
{
  DateTime.Now,
  new DateTime(2013, 02, 13),
  new DateTime(2004, 12, 31)
};
```

Now we can set our `ItemStringFormat` as follows:

```
<ItemsControl
   ItemsSource="{Binding}"
   ItemStringFormat="MMMM dd, yyyy"/>
```

The result is depicted in Figure 8.

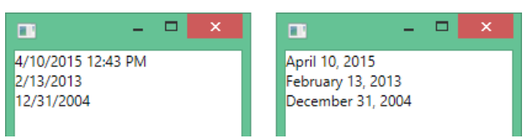


Figure 8 Binding to DateTime instances without ItemStringFormat (left) and with it (right)

As mentioned earlier, we would typically have view models wrapping the data we wish to bind to. Suppose we want to refactor our list of cities into a list of view models representing those cities. We can achieve this very quickly as follows:

```
public sealed class CityViewModel
{
  private readonly string name;
  public CityViewModel(string name)
  {
    this.name = name;
  }
  public string Name => this.name;
}

// in our constructor
this.DataContext = new
List<CityViewModel>
{
  new CityViewModel("London"),
  new CityViewModel("Amsterdam"),
  new CityViewModel("Adelaide")
};
```

If we revert our `ItemsControl` so that it does not specify `ItemStringFormat` and run the application, we see Figure 9. Clearly this is not what we're after. What's happening here is WPF is calling `ToString` on each of our view models in order to obtain a default representation of them. After all, we haven't told WPF that we actually want to show the `Name` property on our view model.

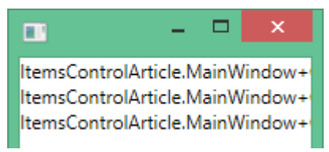


Figure 9 Default visualization of our view models

We can do exactly that by specifying the `DisplayMemberPath` property:

```
<ItemsControl
  ItemsSource="{Binding}"
  DisplayMemberPath="Name"/>
```

This gets us back on track and is visually indistinguishable from Figure 6 where we were using a `List<string>` as our data source. Of course, we can combine `DisplayMemberPath` with `ItemStringFormat`:

```
<ItemsControl
  ItemsSource="{Binding}"
  DisplayMemberPath="Name"
  ItemStringFormat="City: {0}"/>
```

This gets us the same UI as shown in Figure 7.

The view model we created above is pretty pointless. All it does is wrap our city name so it's not adding any value. In reality, we'd likely have several properties for each city:

```
public sealed class CityViewModel :
ReactiveObject
{
  private readonly string name;
  private readonly float population;
  private readonly
  ObservableAsPropertyHelper<IBitmap>
  countryFlag;

  public CityViewModel(string name, float
  population, Task<IBitmap> countryFlag)
  {
    this.name = name;
    this.population = population;
    this.countryFlag = countryFlag
      .ToObservable()
      .ToProperty(this, x =>
        x.CountryFlag);
  }
  public string Name => this.name;
  public float Population => this.
  population;
  public IBitmap CountryFlag => this.
  countryFlag.Value;
}
```

I've added population (in millions) and a country flag image. I'm using Splat for the image so that our view model remains agnostic of the platform on which it is running. You'll notice I'm also deriving from `ReactiveObject` and using something called `ObservableAsPropertyHelper`. These are types from ReactiveUI . ReactiveUI is outside the scope of this article, but you can see that our bitmap is loaded asynchronously. I'm using ReactiveUI as a simple means of surfacing the asynchronously-loaded bitmap as a property. Until it has loaded, our `CountryFlag` returns `null`. Once loaded, a property changed notification is raised for `CountryFlag`, and it returns the loaded bitmap.

I then construct the view models in this manner:

```
new CityViewModel(
  "London",
  8.308f,
  BitmapLoader
    .Current
    .LoadFromResource(
      "pack://application:,,,/
      ItemsControlArticle;component/
      Images/gb.png",
      null,
      null))
```

Again, the details aren't terribly important for the purposes of this article.

Now that we have view models with more interesting data in them, how can we take advantage of this from our view? The `ItemsControl` class includes an `ItemTemplate` property that allows us to specify a rich visual tree to represent each item. Suppose we want to display the city name in bold with the population count underneath it. Off to the right, we want to display the country flag. We can achieve this as follows:

```
<ItemsControl ItemsSource="{Binding}">
 <ItemsControl.ItemTemplate>
  <DataTemplate>
   <Grid>
    <Grid.RowDefinitions>
     <RowDefinition Height="Auto"/>
     <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
     <ColumnDefinition Width="*"/>
     <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <TextBlock
     Text="{Binding Name}"
     FontWeight="Bold"
     FontSize="10pt"/>
    <TextBlock
     Grid.Row="1"
     Text="{Binding Population,
     StringFormat=Population {0:0.#}
     million}"
     FontSize="8pt"
     Foreground="DarkGray"/>
    <Image Grid.Column="1"
     Grid.RowSpan="2"
     Source="{Binding CountryFlag,
     Converter={StaticResource
     ToNativeConverter}}"/>
   </Grid>
  </DataTemplate>
 </ItemsControl.ItemTemplate>
</ItemsControl>
```


What we've effectively done here is told the `ItemsControl` "hey, whenever you need to render an item, please create a copy of this `Grid` with these children and these bindings". The `DataContext` for each `Grid` will be a view model, which is why the bindings will work. If we run the application again, we see the UI in Figure 10. It's far from perfect, but it's a big step forward.

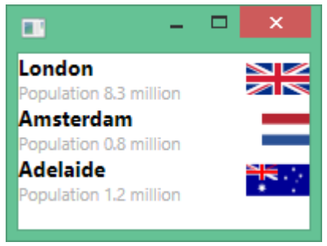


Figure 10 Using an ItemTemplate to customize the visual tree of each item

The `ItemTemplate` gives us a lot of flexibility over how our items are rendered, but `ItemsControl` offers us even more flexibility by way of its `ItemTemplateSelector` property. Setting this property to an instance of `DataTemplateSelector` gives us a means of dynamically selecting a template for each item. Suppose, for example, we generalized our application such that it displays places, not just cities. We can add a `CountryViewModel` alongside our `CityViewModel`. Both view models extend a base view model called `PlaceViewModel`. We would like to display cities differently to countries, but all places are displayed inside the same `ItemsControl`. This is precisely the kind of scenario that `ItemTemplateSelector` accommodates.

Some simple refactoring of our existing code gives us our three view models. We can then define a `DataTemplateSelector` as follows:

```
public sealed class
PlaceDataTemplateSelector :
DataTemplateSelector
{
  public DataTemplate
  CountryDataTemplate { get; set; }
  public DataTemplate CityDataTemplate {
  get; set; }

  public override DataTemplate
  SelectTemplate(object item,
  DependencyObject container)
  {
    if (item is CountryViewModel){

      return CountryDataTemplate;
    }
    else if (item is CityViewModel){
      return CityDataTemplate;
    }
    return null;
  }
}
```

We're using a simple type check to determine which `DataTemplate` to return, where each possible `DataTemplate` is provided to us via a separate property. We can then define an instance of our `PlaceDataTemplateSelector` in the resources for our `Window`:

```
<local:PlaceDataTemplateSelector
x:Key="PlaceDataTemplateSelector">
 <local:PlaceDataTemplateSelector.
 CountryDataTemplate>
  <DataTemplate>
   <Grid>
    <Grid.RowDefinitions>
     <RowDefinition Height="Auto"/>
     <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
     <ColumnDefinition Width="*"/>
     <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <TextBlock
     Text="{Binding Name}"
     FontWeight="Bold"
     FontSize="10pt"/>
    <TextBlock
     Grid.Row="1"
     Text="{Binding Population,
     StringFormat=Population {0:0.#}
     million}"
     FontSize="8pt"
     Foreground="DarkGray"/>
    <Image
     Grid.Column="1"
     Grid.RowSpan="2"
     Source="{Binding CountryFlag,
     Converter={StaticResource
     ToNativeConverter}}"/>
   </Grid>
  </DataTemplate>
</local:PlaceDataTemplateSelector.
CountryDataTemplate>
<local:PlaceDataTemplateSelector.
```

```
 CityDataTemplate>
  <DataTemplate>
   <Grid>
    <Grid.RowDefinitions>
     <RowDefinition Height="Auto"/>
     <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <TextBlock
     Text="{Binding Name}"
     FontWeight="Bold"
     FontSize="8pt"/>
    <TextBlock
     Grid.Row="1"
     Text="{Binding Population,
     StringFormat=Population {0:0.#}
     million}"
     FontSize="8pt"
     Foreground="DarkGray"/>
   </Grid>
  </DataTemplate>
 </local:PlaceDataTemplateSelector.
 CityDataTemplate>
</local:PlaceDataTemplateSelector>
```

We could also inline the definition within our `ItemsControl`, but I usually find it cleaner to separate any relatively complex elements out into the resources section, or even other files altogether. Regardless, we can now modify our `ItemsControl` thusly:

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

The end result is shown in Figure 10. As you can see, we're only showing the flag now if the item represents a country. In addition, the font size for the country name is larger than the font size for city names.

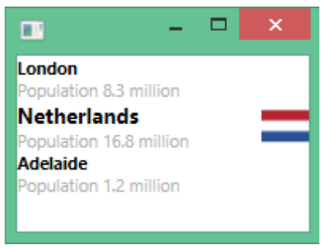


Figure 11 Using ItemTemplateSelector to vary the visuals on a per-item basis

Of course, the flexibility that we get from `ItemTemplateSelector` does not end there. We could have it choose from any number of templates based on any programmable factor we desired. That said, the use of `ItemTemplate` is far more common than `ItemTemplateSelector`. Most of the time you will know at design-time what the item should look like, and there's rarely a need to vary that appearance dynamically.

Conclusion

You should now have a firm grasp on `ItemsControl` fundamentals – how to declare one in XAML, how to populate it with data, and how to customize the appearance of items within it. For some simple scenarios, this is all the knowledge you need to create a solution. However, there is much and more to learn beyond what we've covered in this first part of the article. In part 2 of this article, we'll dig much deeper and acquire the necessary skills to utilize `ItemsControl` in advanced scenarios ■

About the Author

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

INTERNET OF THINGS (IOT), JAVASCRIPT AND AZURE – The Way Ahead

Embracing Open Source Technologies for Connected Devices

With each passing day, we are getting hooked on to an increasing number of small devices. Besides, the Internet drives our lives like never before. It is obvious, as well as natural, that the connectivity of these small devices with the Internet, ultimately, will lead towards their inter-connectivity; where not only data exchange, but decision making will be a shared responsibility of these smart devices. That means the real value of Internet of Things (IoT) does not lie in just home automation, but rather in the data collection by these smart devices and analytics, further up the value chain. In this article, we will explore the possibility of developing applications for IoT devices that capture data from low-cost sensors and communicate with real-time scalable services in Windows Azure – primarily using Open Source Technologies for devices and Managed Services in Azure.

Connected Devices vs People

In the last decade, we have seen multiple platforms (through the power of Internet) facilitate all facets of our communication – email, chat, meetups, networking or career development. People have more than one device that allows them to remain connected to the Internet to access platforms like Facebook, Twitter, WhatsApp, LinkedIn, Meetup, etc. With the current trend, soon the number of Internet connected devices (Internet-of-things, aka IoT) will outnumber the world population. While this increase in the number of Internet-connected devices will increase the network traffic, and will force us to adopt IPv6, it will also open the door to new opportunities for developers, security analysts, business houses and governments.

As developers and architects, it becomes essential for us to think of IoT from the aspects of scale of operation, autonomously connected devices, interoperability between them and seamless collection of data captured by these IoT devices into a centralized data store, that can be used for analytics. This makes the marriage of IoT with the Cloud, perfect!

Internet of Things and Azure – The Way Ahead

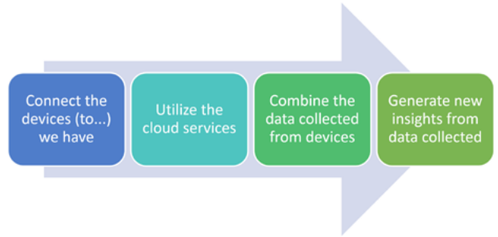
There are several IoT boards available in the market and new ones are getting released every month. Whichever board we choose, the basic process of getting our board connected to the cloud remains the same. The 4 essential steps to have our sensors stream or send data to the cloud are:

1. **Connect the device that we have:** start with connecting sensors to our device and device to the Ethernet/Wireless network.

2. **Utilize the Cloud Services:** build services (Worker Roles, Web Roles, Jobs) that run in the cloud and help in persisting data in one of the data stores like SQL database, DocumentDB, MongoDB, Blobs, etc.

3. **Combine data collected from devices:** With service contracts ready in step 2, we need to ensure that our device consumes the service contracts and is able to push the sensor data to cloud services.

4. **Generate new insights from data collected:** From the network of IoT devices, the collected data can be used to feed Big Data platforms (like HDInsight) for the purpose of analytics.



Typical Tech Stack with Open Source

Choosing the right platform for IoT is an arduous task. The platform, while providing an ease of development, should be extensible to any other IoT board with little modification, should not compromise on the execution speed or security and should execute our code with low-energy factor. Some of the languages that meet the qualifying criteria for IoT platforms are - C, CPP and JavaScript. Choosing a language from these is a matter of personal preference. For this article, we will embrace the open-source technology stack and will explore JavaScript on IoT.

So our tech stack appears like the following:

```
Operating System: Linux / Win10 IoT Core
Hosting Platform: NodeJS, PM2
Language: Javascript / CPP
```

```
Operating System: Windows Server
Hosting Platform: Azure
Language: C#
Database: MS-SQL / Blob / Tables / DocumentDB
```

Whether we use Raspberry Pi, Arduino or Intel Galileo board, the same tech stack can be used with the example illustrated in the sections to follow.

Editorial Note: *For those who want to try this sample on RaspberryPi, please note that Pi cannot read analog input from FS Resistor. Additional analog to digital* converter can be added to original circuit to make this project work. Please see http://acaird.github.io/computers/2015/01/07/raspberry-pi-fsr/ for more information (feedback provided by Shoban Kumar @shobankr).

Smart Garbage Bin

One of the ideas that really fascinates me is the Smart Garbage Bin that some nations have adopted. Smart Garbage Bins tend to decrease the operational cost of monitoring the garbage bins by intelligent monitoring for waste and recyclables. There are different versions of Smart Bins available in market but our rationale (for this article) is -

> ***We need to develop a smart bin that detects if the bin is full and should report to a centralized cloud service. Detection of level of garbage in the bin can be done either by calibrating the weight of garbage or the height of garbage in the bin.***

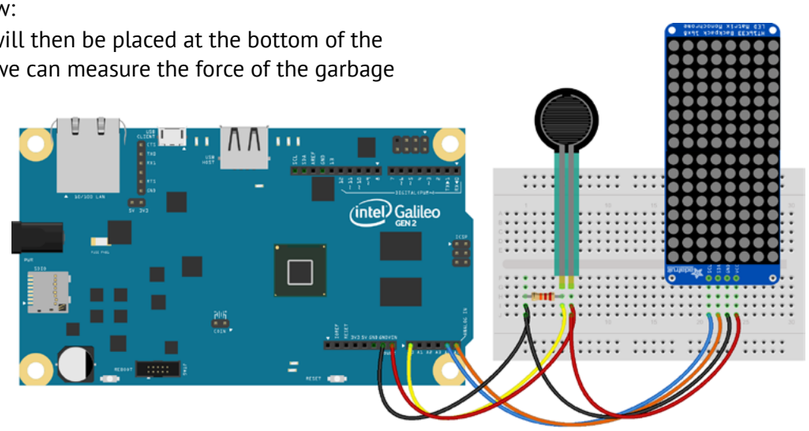
And as mentioned earlier, we will follow the 4 step process to accomplish this.

Setting up the IoT board

To begin with, we need the following electronic components:

- IoT board – let's take Intel Galileo Gen 2 running Yocto (Embedded Linux)
- Force sensitive resistor (FSR) 0.5" or 6" (depending upon surface area of bin)
- Resistor – 22K Ω
- LED Matrix 8x16 HT16K33
- Jumper cables
- Breadboard

The wiring of the Intel Galileo Gen 2 board with FSR sensor and LED matrix needs to be done as shown below:
This setup will then be placed at the bottom of the bin so that we can measure the force of the garbage on the bin.



Setting up the Azure Environment

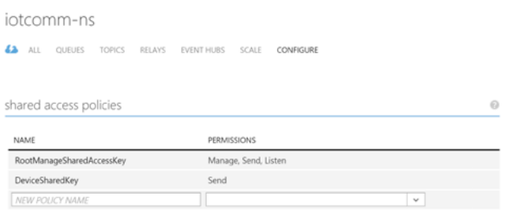
With the tech stack mentioned earlier, we are aiming to have our IoT device code use Javascript/ jQuery to send the sensor data to Azure Cloud Services. From an Azure perspective, we have the option to choose one of the multiple ways available to receive this data, for e.g. service bus queues/ topics, event hubs, blobs, tables, etc. For this application, we will use Azure Service Bus Queues.

So as the next step, we need to create a queue on Azure Management Portal (or using PowerShell scripts) with name `Q.FromSensorRealtimeValue`

Once the queue has been created, we can navigate to the Configure tab and create shared access policy with the following settings:

- Name: DeviceSharedKey
- Permissions: Send

This will restrict the device to only publish messages to the queue.



Now we need a persistent store, like SQL Database, to store messages we receive on this queue.
We need to create a new database using Azure Management Portal and create one table in this new database.

The script of the table is as shown here:

```
CREATE TABLE [dbo].[FSR](
  [ID] [int] IDENTITY(1,1) NOT NULL
  PRIMARY KEY,
  [MeasuredValueF] [numeric](7, 2) NOT
  NULL,
  [DeviceName] [varchar](30) NOT NULL,
  [RecordedAt] [datetime] NOT NULL,
  [ReceivedAt] [datetime] NULL,
  [MessageJSON] [ntext] NULL
)
GO
```

IoT – Cross-Platform, Open Source code

Javascript based platform `NodeJS` gives us the flexibility to write code regardless of any platform, providing us with over thousands of `npm` modules to do almost everything that we can do with managed code like C# and Java. So we can write and build our IoT code using `NodeJS` and then run it on any device / platform. For this application, we will require 4 npm modules to run with `NodeJS` – `os`, `azure`, `galileo-io` and `johnny-five`. If these modules are not installed on your IoT device, we can execute the following installation commands:

```
npm install os
npm install azure
npm install galileo-io
npm install johnny-five
```

Once the modules have been installed on our IoT device, we can paste the following code snippet in a file (say, smartbin.js)

```
var serviceBusConnectionString =
"Endpoint=sb://iotcomm-ns.servicebus.
windows.net/; SharedAccessKeyName=
DeviceSharedKey;SharedAccessKey=
1OD0KM8EjhNmrnlcSTwFfTBQ7xOs9yALpJSwVjP4sIw=";

var os = require("os");
var azure = require('azure');
var Galileo = require("galileo-io");
var five = require("johnny-five");

var serviceBusService =
azure.createServiceBusService
(serviceBusConnectionString);
console.log('Connected to Azure');

var board = new five.Board({
 io: new Galileo()
});

board.on("ready", function () {

 console.log("Connection Established
 with IoT device");

 var fsr = new five.Sensor({
  pin: "A0",
  freq: 60000
 });

 var matrix = new five.Led.Matrix({
  controller: "HT16K33",
  addresses: [0x70],
  dims: "8x16",
  rotation: 2
 });

 var open = [
  "0000000000000000", "0011110000111100",
  "0100001001000010", "1001100110011001",
  "1001100110011001", "0100001001000010",
  "0011110000111100", "0000000000000000",
 ];

 var closed = [
  "0000000000000000", "0000000000000000",
  "0000000000000000", "0000000000000000",
  "0000000000000000", "0000000000000000",
 ];

 var THRESHOLD = 500;
 fsr.scale([0, 1000]).on("data",
 function () {
  if (this.value > THRESHOLD) {
   var message = {
    Hostname: os.hostname(),
    SensorType: "FSR",
    MeasuredValue: this.value,
    RecordedAt: new Date()
   };
   matrix.draw(closed);
   serviceBusService.
```

```
sendQueueMessage("Q.
   FromSensorRealtimeValue", JSON.
   stringify(message), function (error)
   {
    if (!error) {
     console.log("FSR sent to Azure
     Queue");
    }
    else {
     console.log("Error sending to
     Azure" + error);
    }
   }); //sendQueueMessage
  }
  else {
   matrix.draw(open);
  }
 }); //fsr.on("data")
}); //board.on('ready')
```

To deploy this code on our IoT device, we can use FTP tools like FileZilla or use shared folders. Once we have telnet / putty to the device, we can run our IoT `NodeJS` code using

```
node smartbin.js
```

The code will connect to Azure using the connection string and will then initialize the Galileo board using Johnny-Five libraries.

Johnny-Five is Firmata protocol based open-source framework, that can be used to write programs on all Arduino models, Electric Imp, Beagle Bone, Intel Galileo & Edison, Linino One, Pinoccio, pcDuino3, Raspberry Pi, Spark Core, TI Launchpad and more with almost negligible code changes. The library exposes a board object which raises an event 'ready' once the board has been initialized.

Once the board is ready, at a frequency of every 1 minute (60000 millisecond), analog sensor value (i.e. force measured by FSR) on the pin A0 will be determined and the output will be scaled on a range of 0 to 1000.

When the force value goes beyond 500 (i.e. 50% of scaled value 0-1000), it will publish a message to Azure Queue and will dim off the display in LED matrix.

A sample message on the queue will appear:

```
{
 "Hostname": "quark09877",
 "SensorType": "FSR",
 "MeasuredValue": 700,
 "RecordedAt": "Wed Apr 29 2015 18:00:10
 GMT+0000"
}
```

When we expand this solution to run on a network of IoT devices, we should ensure that the time-zones in the devices are either set to UTC and are synchronized, or the Azure Worker Role handles different time-zones.

Azure – Managed code to collect IoT data

To collect the data sent by IoT device, we need a Worker Role process that can pop the message out of the Service Bus Queue and save it in SQL database using Entity Framework. A typical Worker Role requires implementation of 3 methods – `OnStart`, Run and `OnStop`. Our Worker Role process will initiate the queue connection in the `OnStart` method, subscribe to the Queue in Run method and should close the queue connection in `OnStop` method.

```
public override void Run()
{
 Trace.WriteLine("Starting processing of
 messages");
 _queueClient.
 OnMessage((receivedMessage) =>
 ProcessMessage(receivedMessage));
 CompletedEvent.WaitOne();
}
private void
ProcessMessage(BrokeredMessage
receivedMessage)
{
 try
 {
  DateTime receivedAt = DateTime.UtcNow;

  Trace.WriteLine("Processing Service
  Bus message: " +
  receivedMessage.SequenceNumber.
  ToString());
  Stream stream =
  receivedMessage.GetBody<Stream>();

  StreamReader reader =
  new StreamReader(stream);
```

```
string messageBody =
reader.ReadToEnd();

Trace.WriteLine("Message > " +
messageBody);

var sensorMessage = JsonConvert.
DeserializeObject<SensorMessage>
(messageBody);

var sender =
sensorMessage.Hostname.ToUpper();

if (sensorMessage.SensorType ==
SensorType.FSR)
{
 using (var unitOfWork =
 new UnitOfWork())
 {
  decimal measuredValue = -1;
  if (decimal.TryParse(sensorMessage.
  MeasuredValue, out measuredValue))
  {
   var fsr = new FSR();
   fsr.MeasuredValue = measuredValue;
   fsr.DeviceName =
   sensorMessage.Hostname;
   fsr.ReceivedAt = receivedAt;
   fsr.MessageJSON = messageBody;
   fsr.RecordedAt =
   sensorMessage.RecordedAt;
   unitOfWork.Add<FSR>(fsr);
  }
 }

 receivedMessage.Complete();
}
catch (Exception ex)
{
 Trace.TraceError(@"Error saving a FSR
 record due to exception: " +
 ex.ToString());
 receivedMessage.Abandon();
}
}
```

The method `ProcessMessage` gets invoked when a sensor message is received in the queue. The `BrokeredMessage` is designed to support XML serialization, so we have to retrieve the message body as a Stream object and parse it to JSON format using `Newtonsoft.Json` Nuget package. Once we have an object of the sensor data, we can store it in a database (possibly, using Entity Framework or ADO.NET) for analytics purpose.

The code we just saw can be extended to receive data from any sensor attached to any IoT device as far as the device can connect to the Internet.

Beyond reporting measurement

When we are looking forward to building a production-ready IoT device, we would require more efforts in reducing the size of device and ensuring low-battery consumption. For projects like Smart Garbage Bin, we can use solar energy to power our device and we could reduce the frequency of checking the force from 1 minute to 15 minutes.

If we are aiming at creating a network of such IoT devices, we would require more services than just storing this data into database. We can explore several avenues of sending this data to tools like Azure SQL Data Warehouse, Azure Data Lake or Hadoop for trend analysis and then have actions taken based on patterns like peak load time or recycle duration.

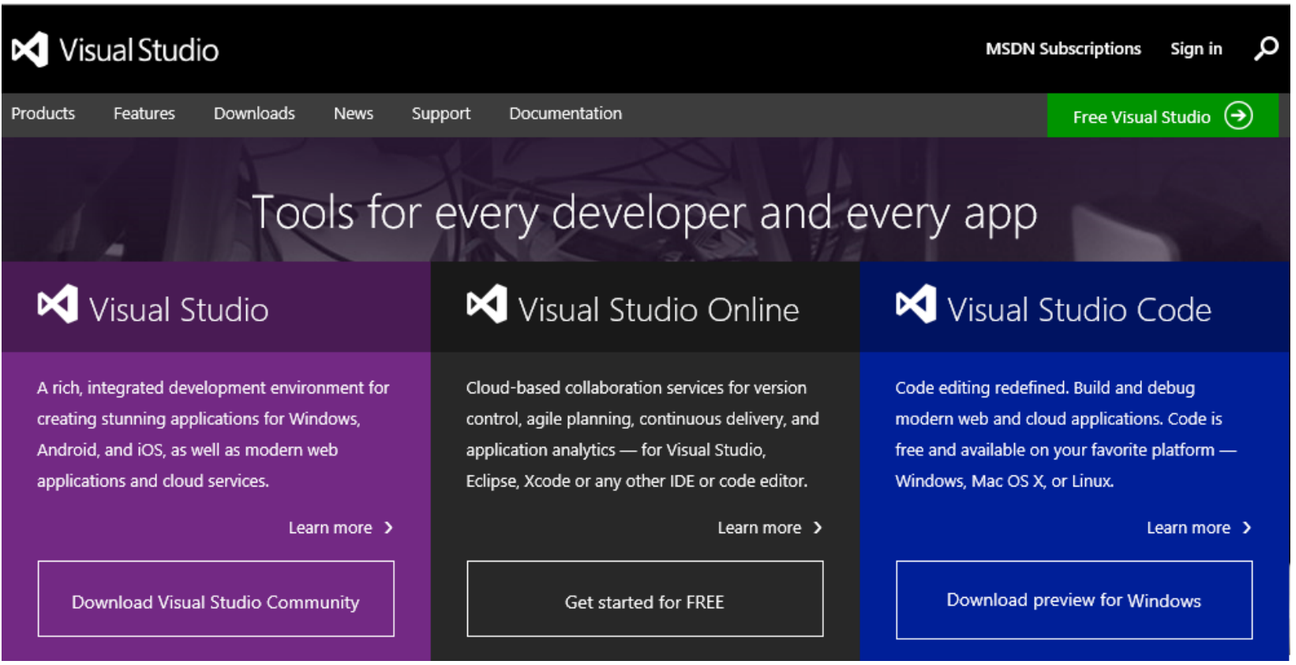
With IoT and Cloud, the possibilities of increasing automation and building smarter homes, cities and nations appear to be seamless! ■

About the Author

punit ganshani

Microsoft® MVP Most Valuable Professional

Punit, a Microsoft .NET MVP and DZone MVB, is the author of 18 technical whitepapers published in DeveloperIQ and a book on C programming. He is an expert at Application Design & Development, Performance Optimization and defining Architecture for hybrid systems involving Microsoft, Open-Source and Messaging Platforms. He is founder of KonfDB platform and runs a blogging platform Codetails, organizes .NET sessions in Singapore, has spoken in various international forums. He maintains his blog at www.ganshani.com



Using REST APIs of TFS and Visual Studio Online

Microsoft has been providing APIs for Team Foundation Services (TFS) from the earliest version of TFS (since Visual Studio 2005). Using these APIs, we can create TFS clients and also write event handlers for the events raised by TFS.

This article is co-authored by Subodh Sohoni and Manish Sharma

In the early days of TFS, we were expected to write only Microsoft .NET clients. We could give reference to the components that contained these APIs, and use the referred classes in the code to access various TFS services. Over the years, to facilitate collaboration between different platforms, we are now also expected to create clients in technologies other than Microsoft.NET; like Java, JavaScript and many others. For these technologies, the components that are written in .NET are not useful.

To assist such programming, Microsoft has recently published the APIs in the form of RESTful services that encapsulate the services of TFS and Visual Studio Online.

In the case of APIs we used in the early versions of TFS, whenever we wanted to access a TFS service programmatically from a client, we had to give reference to assemblies that encapsulated calls to webservices or WCF Services of TFS. It also meant that these assemblies should be present on the computer, the compatible .NET Framework should be installed, and the assemblies should be stored in the Global Assembly Cache. In a nutshell, Team Explorer had to be installed on that computer. These prerequisites were quite restrictive.

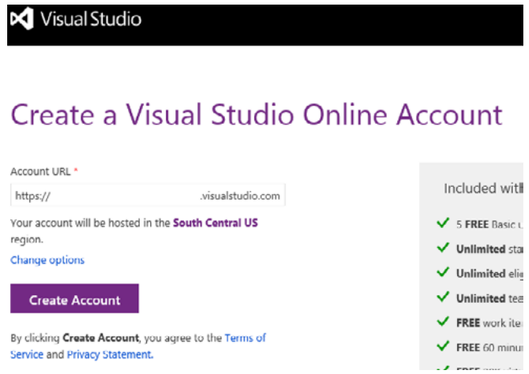
Since we are now dealing with RESTful services, we do not have to worry about these conditions. As long as we can create a HTTP request and send it over the transport to TFS, and are able to read the response; we can write a TFS client. It can be using any technology on any operating system, and on any device. What sounds particularly interesting is a case where a TFS client is able to run on a mobile device.

Accessing Visual Studio Online (VSO) using such a client that uses RESTful services of TFS, has one issue that we have to overcome. That issue is related to security set up by VSO. While accessing through the browser, the Team Web Access application which is an ASP.NET application, uses credentials of the logged-in user. These credentials are usually the users Microsoft account credentials like those derived from Hotmail or Live account. RESTful services of TFS do not support authentication using such credentials. It either supports Basic authentication or OAuth. For each account of VSO, we can enable Basic authentication.

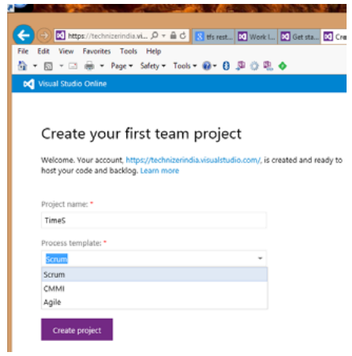
Let us walk through a scenario of creating a VSO account to enable Basic Authentication.

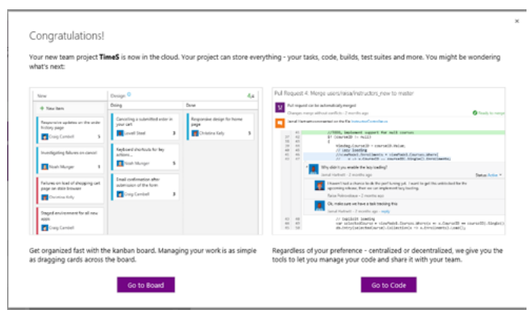
Enabling Basic Authentication

You can create a VSO account by going to http://www.visualstudio.com and then select "Visual Studio Online – Get Started for Free". This free account works for up to 5 users. You will need to login with your Microsoft credentials like Hotmail, Live etc. After that, you should provide a name that should be unique to your account.

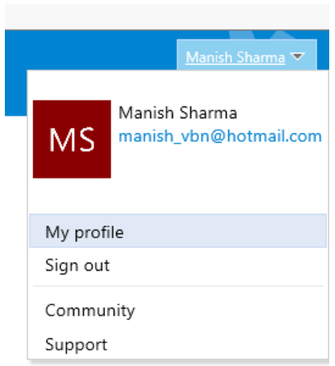


Now you can create your first team project on VSO. You can provide a name to that Team Project and select process template between SCRUM, Agile and CMMI. For our example, we selected the SCRUM process template. That defines the work item types that will be present in the team project. You can also choose the version control mechanism – TFVC or Git.

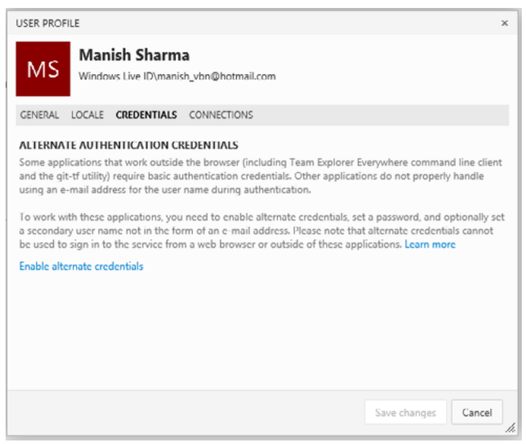




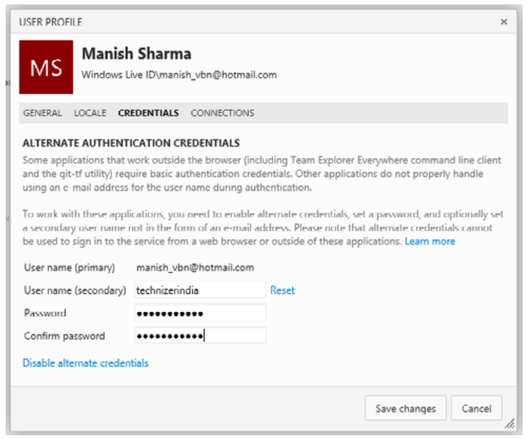
Now that the team project is created, we will go ahead and add the support for Basic authentication in our account. To do so, open the *Settings* section of the account by clicking the name and then *My Profile* on the right top corner. Then in the profile, select the *Credentials* tab.



Now enable Alternate credentials by clicking a link that says 'Enable alternate credentials'.



Then give the User name and Password of your choice. This is the one which will be used for Basic Authentication to your account.



Let us now locate the RESTFul services which we can call to access VSO services. For each account the APIs are available from the base URL https://{account}.visualstudio.com/defaultcollection/_apis/. From here we can access various services of VSO like Projects, Work Item Tracking, Version Control, Test Management, Team Room, Shared Services and Build services. To get URL of each service API, you can visit the page https://www.visualstudio.com/en-us/integrate/api/overview. These services mainly use GET and PATCH methods.

We will focus on one of the most frequently used service and the one that requires most custom client creations, i.e. the Work Item Service. From our custom applications, we often require to create a new work item, view the data stored for that work item and update that work item back in the VSO. Functionality related to Work Item Tracking is available from the services under the URL https://{account}.visualstudio.com/defaultcollection/_apis/wits/workitems.

Let us now view a simple method to get a work item using its ID. Obviously it is a GET method with ID as a querystring parameter. So the UTI will be https://{account}.visualstudio.com/defaultcollection/_apis/wits/workitems?id=1&api-version=1.0

As you must have observed, there is no mention of Team Project name here. That is because work item ids are unique for Team Project Collection. This GET method returns a JSON object with the structure as follows:

```
{
 "count": 3,
 "value": [
  {
   "id": 1,
   "rev": 1,
   "fields": {
    "System.AreaPath": "TimeS",
    "System.TeamProject": "TimeS",
    "System.IterationPath": "TimeS",
    "System.WorkItemType": "Product
     Backlog Item",
    "System.State": "New",
    "System.Reason": "New backlog item",
    "System.CreatedDate":
    "2015-04-29T20:49:20.77Z",
    "System.CreatedBy": "Manish Sharma
     <manish_sharma123@hotmail.com>",
    "System.ChangedDate":
    "2015-05-09T20:49:20.77Z",
    "System.ChangedBy": "Manish Sharma
     <manish_sharma123@hotmail.com>",
    "System.Title": "Customer can sign
      in using their Microsoft Account",
    "Microsoft.VSTS.Scheduling.Effort":
      8,
    "WEF_6CB513B6E70E43499D9FC94E5BBFB784_
     Kanban.Column": "New",
    "System.Description": "Our
      authorization logic needs to allow
      for users with Microsoft accounts
      (formerly Live Ids) - http://msdn.
      microsoft.com/en-us/library/live/
      hh826547.aspx"
   },
   "url": "https://technizer.visualstudio.
   com/DefaultCollection/_apis/wit/
   workItems/1"
  }
}
```

To send this request to the VSO, we will use the instance of the class System.Net.Http.HttpClient

```
HttpClient client = new HttpClient();
```

This client object can create HTTP Requests, Add Headers to that request, Send the request to the known URI and get the Response back.

We will first specify the request header for the type of message as JSON object. Then we specify user credentials to authenticate with VSO. The user credentials are sent in the form of Authorization header with Basic Authentication. The URL will be as mentioned earlier.

The instance of this class can now send a request and it does that asynchronously. That means, we can call Get() method on the URL as GetAsync() which returns a result of call as HttpResponseMessage object.

```
HttpResponseMessage response = client.
GetAsync(Url).Result;
```

To read the contents of that response message, we need to use the *await* keyword.

```
string responseBody = await response.
Content.ReadAsStringAsync();
```

The response is a JSON object with the structure as shown earlier. This response string will have to be typecast into a Work Item object from where we will be able to get the field values. For that, we will write a class that has all the fields or properties that match the JSON object. We created a class called WorkItemDetails for that.

```
public class WorkItemDetails
{
  public string id;  public string rev;
  public IDictionary<string, string>
  fields;
  public string Url;

}
```

So now, we will deserialize the JSON object and typecast it as *WorkItemDetails*. We will use NewtonSoft's JSON.Net package for that. It has a JsonConvert class which can deserialize the JSON object and also typecast it as desired.

```
WorkItemDetails wiDetails =
JsonConvert.DeserializeObject
<WorkItemDetails>(responseBody);
```

ID of the work item is obtained directly. But the fields are returned as a dictionary object. We can get individual fields by using a loop for key-value pair.

```
foreach (KeyValuePair<string, string> fld
in wiDetails.fields)
{
  Console.WriteLine(fld.Key + ":\t" +
  fld.Value);
}
```

The code for this entire functionality will look like the following:

```
static async void GetWorkItem(string
username, string password, int WiId)
{
 try
 {
  using (HttpClient client = new
  HttpClient())
  {
   client.DefaultRequestHeaders.Accept.
   Add(new System.Net.Http.Headers.
   MediaTypeWithQualityHeaderValue
   ("application/json"));
```

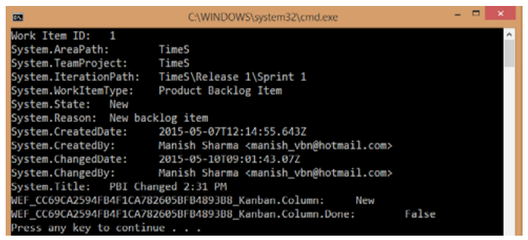
```
client.DefaultRequestHeaders.
Authorization
= new AuthenticationHeaderValue
("Basic",Convert.ToBase64String(
 System.Text.ASCIIEncoding.ASCII.
 GetBytes(
  string.Format("{0}:{1}", username,
  password))));

string Url = "https://ssgsonline.
visualstudio.com/defaultcollection/
_apis/wit/workitems?id=
1&api-version=1.0";

using (HttpResponseMessage response =
client.GetAsync(Url).Result)
{
 response.EnsureSuccessStatusCode();
 string responseBody = await
 response.Content.
 ReadAsStringAsync();
 WorkItemDetails wiDetails =
 JsonConvert.DeserializeObject
 <WorkItemDetails>(responseBody);
 Console.WriteLine("Work Item ID:
 \t" + wiDetails.id);

 foreach (KeyValuePair<string,
 string>
 fld in wiDetails.fields)
 {
  Console.WriteLine(fld.Key + ":\t" +
  fld.Value);
 }
 }
}
Catch (Exception ex)
{
 Console.WriteLine(ex.Message);
}
}
```

The output looks similar to the following:



To create a new work item, we will use the PATCH method for this service. PATCH method of this service accepts a parameter that indicates work item of which work item type is to be created. The URL of the method that creates a Task looks like this:

https://{account}/DefaultCollection/{TeamProject}/_apis/wit/workitems/$Task?api-version=1.0

In addition to that, this method accepts following data in the JSON format:

```
[
  {
    "op": "add",
    "path": "/fields/System.Title",
    "value": "JavaScript implementation
    for Microsoft Account"
  }
]
```

The value of variable "op" indicates that the field is to be added, "path" is the name of the field to be given the value and "value" obviously is the value to be given to that field. It has to be sent as collection of fields. The code for creation of work item will look like this:

```
static async void CreateWorkItem(string
username, string password)
{
 try
 {
  using (HttpClient client = new
  HttpClient())
  {
   client.DefaultRequestHeaders.Accept.
   Add(new System.Net.Http.Headers.
   MediaTypeWithQualityHeaderValue
   ("application/json-patch+json"));

   client.DefaultRequestHeaders.
   Authorization = new
   AuthenticationHeaderValue
   ("Basic",Convert.ToBase64String(
   System.Text.ASCIIEncoding.ASCII.
   GetBytes(
   string.Format("{0}:{1}", username,
   password))));

   WorkItemPostData wiPostData = new
   WorkItemPostData();

   wiPostData.op = "add";
   wiPostData.path =
   "/fields/System.Title";
   wiPostData.value = "Employee edits
   other employees profile";

   List<WorkItemPostData> wiPostDataArr
   = new List<WorkItemPostData> {
   wiPostData };
   string wiPostDataString =JsonConvert.
   SerializeObject(wiPostDataArr);
   HttpContent wiPostDataContent = new
   StringContent(wiPostDataString,
   Encoding.UTF8, "application/json-
   patch+json");

   string Url =
   "https://ssgsonline.visualstudio.com/
   DefaultCollection/SSGS EMS SCRUM/_
   apis/wit/workitems/
   $Product%20Backlog%20Item?api-
   version=1.0";

   using (HttpResponseMessage response =
   client.PatchAsync(Url,
   wiPostDataContent).Result)
   {
    response.EnsureSuccessStatusCode();
    string ResponseContent = await
    response.Content.
    ReadAsStringAsync();
   }
  }
 }
 catch(Exception ex)
 {
  Console.WriteLine(ex.ToString());
  Console.ReadLine();
 }
}
```

There are two tasks that we have to do in this case. One is to create a class that represents the data to be sent to PATCH the method.

```
public class WorkItemPostData
{
  public string op;
  public string path;
  public string value;
}
```

Second is that the HttpClient does not by default support the PATCH method. For that, we have to create an extension method ref https://msdn.microsoft.com/en-IN/library/bb383977.aspx. This extension method is called PatchAsync() and has the following code:

```
public async static
Task<HttpResponseMessage>
PatchAsync(this HttpClient client,
string requestUri, HttpContent content)

{
  var method = new HttpMethod("PATCH");
  var request = new
  HttpRequestMessage(method, requestUri)
  {
    Content = content
  };
  return await client.
  SendAsync(request);
}
```

To update the work item, we have to use the same PATCH method but with different parameter. We have to now send the work item id as the parameter. The URL for the same is:

https://{account}.visualstudio.com/defaultcollection/_apis/wit/workitems/{id}?api-version={version}

The data that will be accepted, is in the JSON format as shown here:

```
[
  {
    "op": "replace",
    "path": { string }
    "value": { string or int, depending
    on the field }
  }
]
```

The "op" variable has the values "add", "replace", "remove" and "test". The last one i.e. "test" checks if the operation can be performed successfully or not; it does not save the work item actually.

Since we are sending a collection of fields, all of them can be updated in one round trip to server. The code will look quite similar to the method for creating work item but with few differences:

```
static async void UpdateWorkItem(string
username, string password)
{
 try
 {

  using (HttpClient client = new
  HttpClient())
  {
```

```
client.DefaultRequestHeaders.Accept.
Add(new System.Net.Http.Headers.
MediaTypeWithQualityHeaderValue
("application/json-patch+json"));

client.DefaultRequestHeaders.
Authorization = new
AuthenticationHeaderValue("Basic",
Convert.ToBase64String(

System.Text.ASCIIEncoding.ASCII.
GetBytes(string.Format("{0}:{1}",
username, password))));

WorkItemPostData wiPostData =
new WorkItemPostData();

wiPostData.op = "replace";

wiPostData.path =
"/fields/System.Title";
wiPostData.value = "Employee edits
own profile in broser based app";

List<WorkItemPostData> wiPostDataArr
= new List<WorkItemPostData> {
wiPostData };

string wiPostDataString=JsonConvert.
SerializeObject(wiPostDataArr);

HttpContent wiPostDataContent
= new StringContent(wiPostDataString,
Encoding.UTF8,
"application/json-patch+json");

string Url =
"https://ssgsonline.visualstudio.com/
DefaultCollection/_apis/wit/
workitems/1?api-version=1.0";

using (HttpResponseMessage response =
client.PatchAsync(Url,
wiPostDataContent).Result)
{
 response.EnsureSuccessStatusCode();
 string ResponseContent = await
 response.Content.
 ReadAsStringAsync();
}
}
}

catch(Exception ex)
{
 Console.WriteLine(ex.ToString());
 Console.ReadLine();
}
}
```

The only two differences are as follows:

1. Variable "op" has value "replace" as we are replacing the title of work item

2. URL now changes so that team project name is removed and instead of work item type we are sending the ID of work item to be updated.

So far we have been using BASIC authentication which is useful for applications like Console Application, Windows Desktop (WinForms) application, Windows Store Applications, other technology applications and can also be used for ASP.NET web applications. One issue in Basic authentication is that password is sent over the network in plain text. Although we are using a secure protocol like https, it still is less secure compared to any other authentication mechanism that does not require us to send any password over the internet. One such mechanism is OAuth.

Using OAuth

OAuth is a service for authentication and authorization (delegation based authorization) which follows open standards. Open Standard is something which is freely adopted, can be freely used and implemented, and can be extended.

OAuth allows an application to act on user's behalf to access server resources without providing their credentials. OAuth Protocol is designed to work with HTTP, also OAuth allows web users to log into third party sites using their available accounts.

Let's understand Oauth through a simple example.

Suppose you are an end user for a website which provides contacts service, so that you can store contacts and later use them to send mails.

Hypothetically, say I wrote an application which can send greetings to email addresses. You want to use this application. Now my Application will use your mailing list to send greetings every morning as per scheduled. To do so, my application needs access to the website or the server which has your contacts stored.

The application I have developed can access the email address available on the server with your credentials. If you are willing to share those credentials, I am more than happy to accept them. But I am sure, you will never share your credentials with me or in fact with any third party application you are using.

So what's the solution? This is the scenario where OAuth can help us. It uses one more party, the known and trusted authorization engine. Let us now step through the entire scenario.

You register with my application and then give me authority to send a request to your authentication provider, so that it will authenticate you and allow me to use some token as evidence of that. This is a one time task that you need to do.

Step 1: When you access my application, you are redirected to that authorization engine. You can get authorization using that trusted authorization engine, where I have no access to your credentials data.

Step 2: That engine vouches for your authenticity. It gives me some token as evidence of your authenticity.

Step 3: I will take that token as evidence of your authentication and send it with my request to the contacts server.

Step 4: That token will be acceptable to your contacts server as evidence of your authenticity and also your trust in my application. It will then provide your contact list to my application.

Hope this helps you with some OAuth concepts.

Let's talk about the main components in an OAuth communication. So in accordance with the example used above:

1. End User – You

2. Resource – Contacts Server, it is your Resource Provider.

3. Consumer – Application created by me, it is Resource Consumer

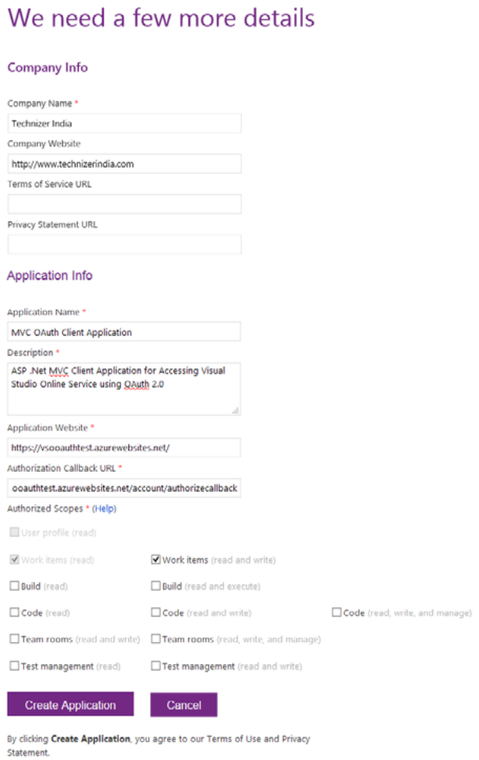
4. OAuth Provider - Trusted OAuth Providers such as Microsoft

Let us now extend this example to include VSO which will be the Resource Provider. I have written an application that gives you a simple interface to view work items, create new work items and edit existing work items. It does so on VSO using RESTFul API of VSO. So you need to allow access to my application, so that it can access the VSO on your behalf, without providing your credentials to me or my application. My application demands OAuth Token issued by the trusted authorization engine, which in this case is Microsoft's VSO Authorization Service.

So if my application wants to use OAuth, there are some steps needed:

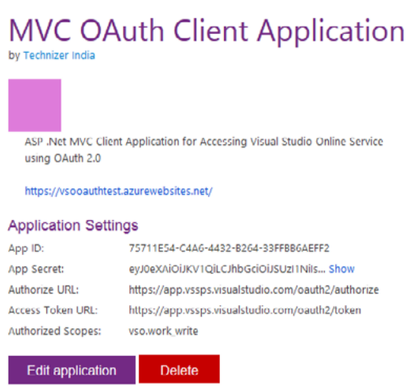
1. First step is to register my application (Consumer) to the OAuth Server (Provider). Remember this provider also supports your resource (VSO).

2. When I register my application, I have to specify what all resources or features my application will access. These are also known as Scopes. I can restrict the scope to view work items only or allow view, create and edit work items. Along with these details, I also have to specify the Redirect URL. In the case of regular user accessing this application in future, this URL will be used by VSO Authorization Service to redirect the user to this URL.



3. Once I complete the above steps, my application will be registered with the provider and provider will provide me the following details:

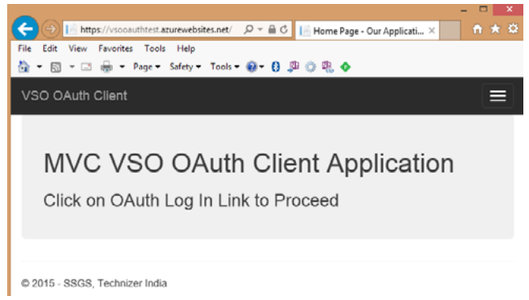
a. App ID – Unique ID for my application
b. Secret Key – Also known as App Secret, will we used by application



4. Once we have got the details, we will use this information in the application program to get OAuth Token from the provider on behalf of end user, and use the token to Log In and access the VSO.

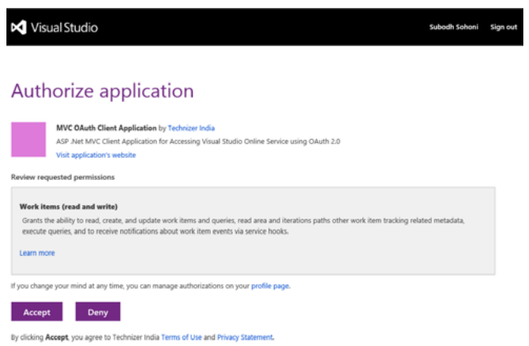
Following are the steps taken between End User, Provider and Consumer to complete the process:

1. End User browses the Consumer application (Consumer).



2. Consumer Application redirects the request to Authorization Server (Provider), passing the App Id, Scope and Redirect URL, through the User Agent (browser). In the application we used a login hyperlink which opens the page that has the following code:

```
var authorizeUrl = String.
Format("https://app.vssps.
visualstudio.com/oauth2/
authorize?client_id={0}&response_
type=Assertion&state=TestUser&scope=vso.
work_write&redirect_uri={1}",
HttpContext.Application["AppId"].
ToString(),
HttpContext.Application["RedirectUrl"].
ToString());
return new RedirectResult(authorizeUrl);
```



3. If the user is not logged in to the Authorization Server (Provider), she/he has to Log In and Authorize Consumer application to access resources. The user also has a choice to deny the access.

4. If user allows the access, a single use authorization code is generated and given back to the Redirect URL specified by the consumer Application. In our application, AuthorizeCallback is the method that accepts mentioned callback with authorization code.

```
public async Task<ActionResult>
AuthorizeCallback(string code, string
state)
{
  var token = await GetToken(code,
  false);
  Session["TokenTimeout"] = DateTime.
  Now.AddSeconds(Int32.Parse(token.
  expires_in)); //Token is Valid for
  approx. 14 Mins
  Session["AuthToken"] = token;

  return RedirectToAction("GetWorkItems",
  "Home");
}
```

5. Now Consumer Application passes the authorization code, its own App Id and secret to the authorization server (Provider) and also the Redirect URL where user will receive the OAuth Token. (This token may be permanent or timestamp based, which is valid for a duration)

```
public async Task<AccessToken>
GetToken(string code, bool refresh)
{
  string tokenUrl = "https://app.vssps.
  visualstudio.com/oauth2/token";
  string appSecret = HttpContext.
  Application["AppSecret"].ToString();
  string redirectUrl = HttpContext.
  Application["RedirectUrl"].ToString();
  string urlData = string.Empty;

  if (refresh)
  {
    urlData = string.Format("client_
    assertion_type=urn:ietf:params:
    oauth:client-assertion-type:jwt-
    bearer&client_assertion={0}
    &grant_type=refresh_token&
    assertion={1}&redirect_uri={2}",
      Uri.EscapeUriString(appSecret),
      Uri.EscapeUriString(code),
      redirectUrl);
  }
  else
  {
    urlData = string.Format("client_
    assertion_type=urn:ietf:param
    s:oauth:client-assertion-type:jwt-
    bearer&client_assertion={0}&grant_
    type=urn:ietf:params:oauth:grant-
    type:jwt-bearer&assertion={1}
    &redirect_uri={2}",
      Uri.EscapeUriString(appSecret),
      Uri.EscapeUriString(code),
      redirectUrl);
  }

  string responseData = string.Empty;
  AccessToken oauthToken = null;

  HttpWebRequest request=
  (HttpWebRequest)WebRequest.
  Create(tokenUrl);

  request.Method = "POST";
  request.ContentType = "application/
  x-www-form-urlencoded";

  using (StreamWriter sw =
  new StreamWriter(await request.
  GetRequestStreamAsync()))
  {
    sw.Write(urlData);
  }

  HttpWebResponse response =
  (HttpWebResponse)(await request.
  GetResponseAsync());

  if (response.StatusCode ==
  HttpStatusCode.OK)
  {
    using (StreamReader
    srResponseReader =
    new StreamReader
    (response.GetResponseStream()))
    {
      responseData = srResponseReader.
      ReadToEnd();
    }

    oauthToken = JsonConvert.
    DeserializeObject<AccessToken>
    (responseData);
    }

    return oauthToken;
  }
```

6. After Validating the details, Authorization Server (Provider) returns an access token i.e. OAuth Token to Consumer Application (oauthToken from our earlier code).

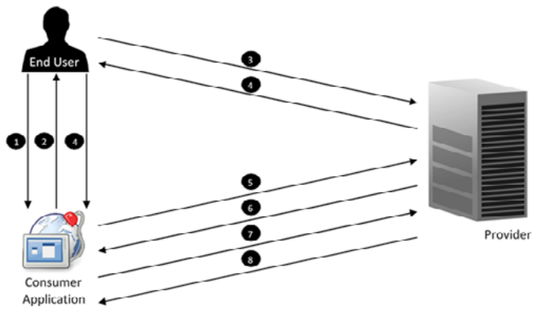
7. Consumer application uses this token to access resources on behalf of the user, Consumer App has to send the token with every request to the Resource Provider, which in our case is VSO

```
[HttpPost]
public ActionResult
Create(EntityForCreateAndUpdate c)
{
  if (DateTime.
  Parse(Session["TokenTimeout"].
  ToString()) <= DateTime.Now)
  return RedirectToAction("RefreshToken",
  "Account", new { url = Request.Url.
  LocalPath });

  if (ModelState.IsValid)
  {
    ViewBag.Op = "Created";
    var t = helper.
    CreateWorkItemAsync(((AccessToken)
    Session["AuthToken"]),
    c.WorkItemTitle);
    t.Wait();
    return View("Details", t.Result);
  }
  else
    return View();
  }
```

8. Resource Provider validates the token and returns the resources needed by the Consumer Application. In this code, we created a new work item in our team project.

These steps will be easier to understand using the following visual:



Some points to remember:

1. The current version of OAuth is OAuth 2.0

2. OAuth 2.0 Tokens can be shared only on secured channel i.e. HTTPS. It relies on SSAL to provide encryption.

3. Facebook, Google, Twitter and Microsoft Servers are some of the OAuth Providers.

Conclusion

Visual Studio Online and TFS 2015 (RC Onwards) provides RESTful APIs that allow you to extend the functionality of VSO from your apps and services. The possibilities of integration via client applications using any platform or device are endless; right from iOS, Node.js, Android and our very own Windows ■

Download the entire source code from GitHub at / bit.ly/dncm19-vsotfsrestapi

• • • • • • •

About the Authors

subodh sohoni

Microsoft® Most Valuable Professional

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

SINGLE RESPONSIBILITY PRINCIPLE

(Software Gardening: Seeds)

In the last edition (May-June 2015 edition) of the DNC Magazine, I covered some basics of Object Oriented Programming (OOP). As a quick review, I discussed different types of inheritance, polymorphism, encapsulation, loose coupling, and tight cohesion. Now I want to dive deeper into good OOP techniques and begin a discussion on SOLID.

First introduced by Robert "Uncle Bob" Martin, SOLID is not new. Uncle Bob simply took concepts that had been around for years and put them together. However, he didn't have them in SOLID order. We can credit Michael Feathers for coming up with the SOLID acronym. So, what is SOLID? Well, it is five OOP principles, the first letter of each spelling out SOLID: **S**ingle Responsibility, **O**pen/Closed, **L**iskov Substitution, **I**nterface Segregation, and **D**ependency Inversion. Over the next five issues, I'll cover each one of these concepts. While originally targeting OOP, many of these concepts apply to non-OOP languages as well.

The **S**ingle Responsibility Principle (SRP) states that a class should do one thing and one thing only. After years of working with OOP code, I've found that many developers violate this principle all the time. Yes, we write classes and methods, but we tend to write one big method that does something in a procedural manner rather than having smaller classes that do one thing. Here's some typical code that demonstrates this.

```
public class CsvFileProcessor
{
 public void Process(string filename)
 {
  TextReader tr = new
  StreamReader(filename);
  tr.ReadToEnd();
  tr.Close();

  var conn = new
  SqlConnection("server=(local);
  integrated security=sspi;
  database=SRP");
  conn.Open();

  string[] lines = tr.ToString().
  Split(new string[] {@"\r\l"},
  StringSplitOptions.RemoveEmptyEntries);
  foreach( string line in lines)
  {
   string[] columns = line.Split(new
   string[] {","}, StringSplitOptions.
   RemoveEmptyEntries);
   var command = conn.CreateCommand();
   command.CommandText = "INSERT INTO
   People (FirstName, LastName, Email)
   VALUES (@FirstName, @LastName,
   @Email)";
   command.Parameters.AddWithValue("@
   FirstName", columns[0]);
   command.Parameters.AddWithValue("@
   LastName", columns[1]);
   command.Parameters.AddWithValue("@
   Email", columns[2]);
   command.ExecuteNonQuery();
  }
  conn.Close();
 }
}
```

How many things is this class doing? One? Two? Three? More? You may be tempted to say one. That is, the class processes a CSV file. Look at this class another way. How would you unit test this? It wouldn't be easy. What if you had other things like data validation and error logging? How would you unit test it then?

The truth is, this class is doing three things:

1. Reading a CSV file
2. Parsing the CSV file
3. Storing the data

Doing lots of things in a class is bad not just because it is difficult to unit test, but it increases the odds of introducing bugs. If you change the code in the Parsing section, and you add a bug, then Reading and Storing are also broken. And, because unit tests will not exist or are very complex, it also takes longer to track down and fix the bug.

In order to fix this, we need to break down the code into the individual pieces. You may be thinking you can just have three methods, one for each piece of functionality. But go back to the definition of SRP. It says that a class should have only one purpose. So, we need three classes to do the work. Alright, we'll actually have more as you'll see in a moment.

The way to fix this code is through code refactoring. Initially, we'll put each piece of functionality into its own method.

```
public class CsvFileProcessor
{
  public void Process(string filename)
  {
    var csvData = ReadCsv(filename);
    var parsedData = ParseCsv(csvData);
    StoreCsvData(parsedData);
  }

  public string ReadCsv(string filename)
  {
    TextReader tr = new
    StreamReader(filename);
    tr.ReadToEnd();
    tr.Close();
    return tr.ToString();
  }

  public string[] ParseCsv(string
  csvData)
  {
    return csvData.ToString().
    Split(new string[] { @"\r\l" },
    StringSplitOptions.
    RemoveEmptyEntries);
  }

  public void StoreCsvData(string[]
  csvData)
  {
    var conn = new
    SqlConnection("server=(local);
    integrated security=sspi;
    database=SRP");
    conn.Open();
    foreach (string line in csvData)
```

```
    {
      string[] columns = line.Split(new
      string[] { "," },
      StringSplitOptions.
      RemoveEmptyEntries);
      var command = conn.
      CreateCommand();
      command.CommandText =
      "INSERT INTO People (FirstName,
      LastName, Email) VALUES
      (@FirstName, @LastName, @Email)";
      command.Parameters.AddWithValue("@
      FirstName", columns[0]);
      command.Parameters.AddWithValue("@
      LastName", columns[1]);
      command.Parameters.AddWithValue("@
      Email", columns[2]);
      command.ExecuteNonQuery();
    }
    conn.Close();
  }
}
```

As you can see, things still aren't quite right. We're parsing the CSV file into rows in the ParseCsv() method, but additional parsing is happening in the StoreCsvData() method to get each row into columns. The way to fix that is with a *ContactDTO* that stores the data from each row.

The next step is to add the DTO, but I'll skip a step and also break out each method into its own class. But I'm going to think ahead here too. What if the data doesn't come in as CSV? What it its XML or JSON or something else? You solve this with *interfaces*.

```
public interface IContactDataProvider
{
  string Read();
}

public interface IContactParser
{
  IList<ContactDTO> Parse(string
  contactList);
}

public interface IContactWriter
{
  void Write(IList<ContactDTO>
  contactData);
}

public class ContactProcessor
{
  public void
  Process(IContactDataProvider cdp,
  IContactParser cp, IContactWriter cw)
  {
    var providedData = cdp.Read();
    var parsedData =
    cp.Parse(providedData);
    cw.Write(parsedData);
  }
}

public class CSVContactDataProvider :
IContactDataProvider
{
  private readonly string _filename;

  public CSVContactDataProvider(string
  filename)
  {
    _filename = filename;
  }

  public string Read()
  {
    TextReader tr = new StreamReader(_
    filename);
    tr.ReadToEnd();
    tr.Close();
    return tr.ToString();
  }
}

public class CSVContactParser :
IContactParser
{
  public IList<ContactDTO> Parse(string
  csvData)
  {
    IList<ContactDTO> contacts = new
    List<ContactDTO>();
    string[] lines = csvData.
    Split(new string[] { @"\r\l" },
    StringSplitOptions.
    RemoveEmptyEntries);
    foreach (string line in lines)
    {
      string[] columns = line.Split(new
      string[] { "," },
      StringSplitOptions.
      RemoveEmptyEntries);
      var contact = new ContactDTO
      {
        FirstName = columns[0],
        LastName = columns[1],
        Email = columns[2]
      };
      contacts.Add(contact);
    }
    return contacts;
  }
}

public class ADOContactWriter :
IContactWriter
{
  public void Write(IList<ContactDTO>
  contacts)
  {
    var conn = new
    SqlConnection("server=(local);
    integrated security=sspi;
    database=SRP");
    conn.Open();
    foreach (var contact in contacts)
    {
      var command = conn.
      CreateCommand();
      command.CommandText = "INSERT INTO
      People (FirstName, LastName,
      Email) VALUES (@FirstName,
      @LastName, @Email)";
      command.Parameters.AddWithValue("@
      FirstName", contact.FirstName);
      command.Parameters.AddWithValue("@
      LastName", contact.LastName);
      command.Parameters.AddWithValue("@
      Email", contact.Email);
      command.ExecuteNonQuery();
    }
    conn.Close();

  }
}

public class ContactDTO
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Email { get; set; }
}
```

We're using generic method names of Read, Parse, and Write because we don't know what type of data we'll get. Now we can easily unit test this code. We can also easily modify the Parse code and if we introduce a new bug, it won't affect the Read and Write code. Another bonus is that we've loosely coupled the implementation.

So, there you have it. We took what is fairly common procedural code and refactored it using the Single Responsibility Principle. Next time you look at a class, ask yourself if you can refactor it to use SRP. Applying the S of SOLID will help your code to be green, lush, and vibrant, and you're on your way to having a software garden.

About Software Gardening

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

About the Author

craig berntson

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.

Build a WINDOWS 10 UNIVERSAL APP - First Look

Unless you have been living under a rock for the last couple of months, you probably already know that Microsoft's next version of Windows will be called Windows 10 and it will be available from July 29th 2015 onwards. I have been using the Windows 10 preview version *in my Windows Phone and Surface and I am really excited for the final launch. As a passionate developer, I would encourage every fellow developer to try out the preview version and experience why it is the right time be a Microsoft Developer.*

In this article, we will see what's new in Windows 10 and as a developer, how you can make use of new features to build apps, or even turn your existing Websites to Apps (Yes! You heard it right) and use device features like Calendar, Notifications etc. At the end of the article, we will also develop a simple Universal App to see how it behaves in a Tablet/Desktop and Windows Phone.

Why is Windows 10 different?

One OS

To counter the erosion of its once impregnable desktop market and the growing popularity of Linux servers, Microsoft's strategy is very simple – run one version of Windows on everything. So unlike previous versions of Windows, Windows 10 will be a single operating system that will run across all devices starting from High End desktops and laptops, all the way through Tablets and Smart Phones, and even on IoT devices like Raspberry Pi 2, Intel Galileo etc.

As a developer, this means there are no more multiple packages and versions. You can maintain one code base and develop apps that run across a large number of devices running Windows 10.

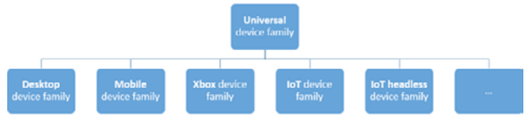
One developer platform

We are already familiar with **Universal apps** which allows us to share code between Windows 8.1 app and Windows Phone app. Windows 10 introduces a completely new different way of developing apps targeting multiple devices and it is called **UWP apps** or **Universal Windows Platform apps**.

This new platform also allows us to easily activate/deactivate features, in different devices, using unique APIs that target them. This also means there is only **One Dev Centre** to manage our apps and **One Store** where users browse and download our apps.

Multiple device families

Windows 10 UWP apps target different variety of devices and they are grouped into **Device families** as shown below.



Grouping these devices makes it easier for us to identify unique APIs targeting a particular type of device. Developers can enable their apps to run in one, or all of the devices and use adaptive code to activate or deactivate features.

Later in this article, we will see how we can run a sample app in Windows 10 Tablet, Windows Desktop and Windows Phone.

API Contracts

API Contracts allow us to check the availability of a Windows feature during runtime. This makes it possible to use device specific unique features and provide a different experience for the user in different devices, but maintain the same code.

Adaptive UX

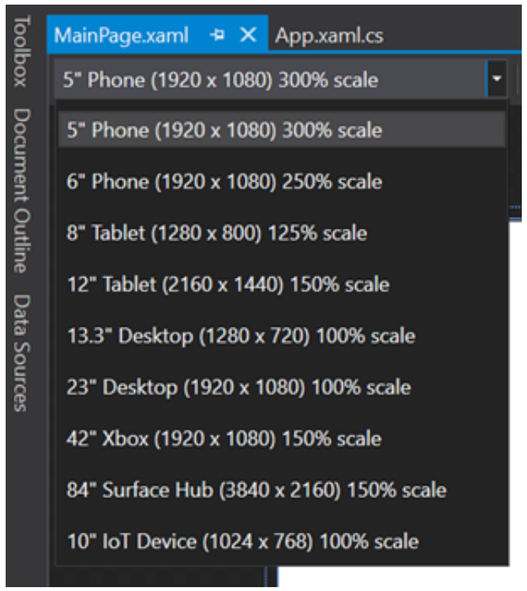
Windows 10 allows us to develop apps that use a single UI that can adapt itself in small or large screens, *without writing any code behind*. The new **RelativePanel** makes this easy for developers to implement layouts which is based on relationship between its child elements. This also means lesser

and cleaner XAML code.

Adaptive Visual States allow us to change the UI based on the changes in size of the window *without writing any extra code*.

New controls like calendar, split view etc. have been introduced and existing controls have been updated to work well in different screens.

Device Preview toolbar in Visual Studio allows us to preview the UI in different devices without running the app. Here is a screenshot of the toolbar.



Adaptive Scaling makes it easy to reuse assets from other operating system projects like Android and iOS which will reduce a lot of design time. **Common Input Handling** makes it even easier to gather input from various sources like Touch, Keyboard, Xbox controller etc. with only a few lines of code.

Hosted Web Apps

If you are a Web Developer, you can convert your existing Web Apps to a Universal Windows Apps using this model and even use universal APIs like camera, calendar, contact list etc. Users will be able to download your App like any other app from the store. Any updates done to the website is reflected immediately in the app. Hosted Web Apps can also use *Cortana* to have unique user experience in their website.

If your website does not have a Windows App, then this may be a good starting point with very less code, time and investment.

Cortana

Cortana is now a part of Windows 10 and is more open for developers. Our App (even Hosted Web App) can now react to voice commands and even override the default app behaviour (Imagine developing a Weather App which will be the default weather app for Cortana)

Reference Links

Here are some links for you to get started with Windows 10

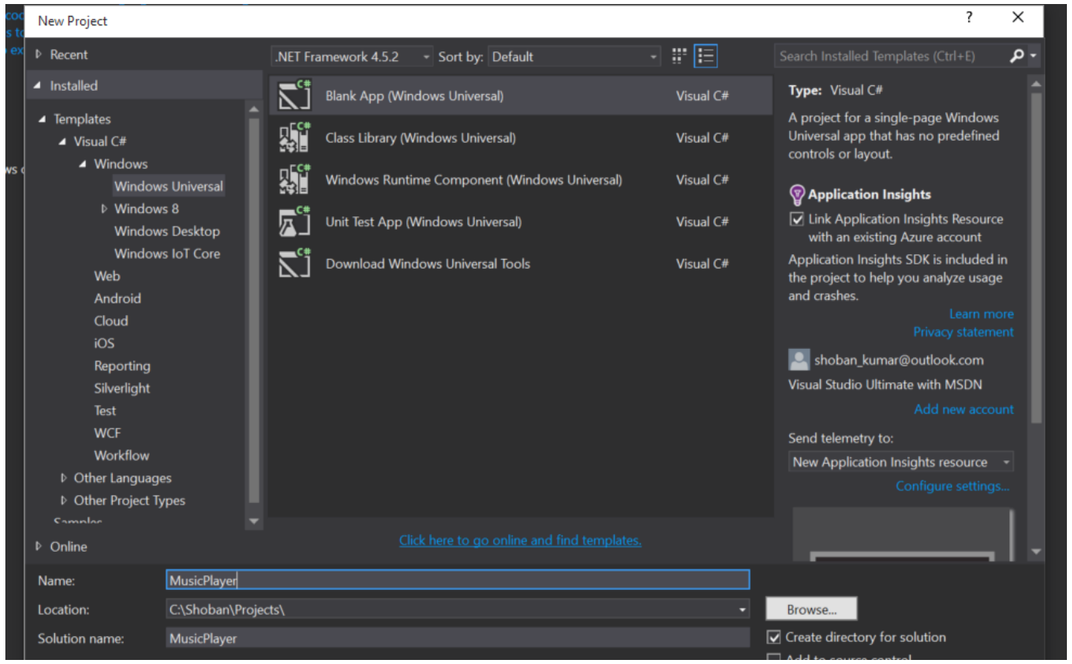
1. Windows 10 Inside Preview
2. Download Windows 10 Mobile Insider Preview
3. Visual Studio 2015 RC
4. Microsoft Virtual Academy – A Developer's Guide to Windows 10

Sample Windows 10 App

Let us develop a simple Music Player app to see how the Adaptive UI works in different devices without writing any extra C# code.

Note: *Since this is an introduction article, we are covering just one feature. In the forthcoming articles, we will be covering the others.*

Step 1: Fire up Visual Studio 2015 RC, create a new **Windows Universal Blank App** and name it MusicPlayer.



Step 2: Add the following XAML to the main Grid in MainPage.xaml

```
<VisualStateManager.VisualStateGroups>
 <VisualStateGroup
x:Name="VisualStateGroup">
  <VisualState x:Name="narrowView">
   <VisualState.StateTriggers>
    <AdaptiveTrigger MinWindowWidth="0"
    />
   </VisualState.StateTriggers>

   <VisualState.Setters>
    <Setter Target="sldProgress.
    Visibility" Value="Collapsed"/>
    <Setter Target="txtEnd.Visibility"
    Value="Collapsed" />
    <Setter Target="txtStart.Visibility"
    Value="Collapsed" />
   </VisualState.Setters>
  </VisualState>

  <VisualState x:Name="wideView">
   <VisualState.StateTriggers>
    <AdaptiveTrigger
    MinWindowWidth="1000" />
   </VisualState.StateTriggers>

   <VisualState.Setters>
    <Setter Target="sldProgress.
    Visibility" Value="Visible"/>
    <Setter Target="txtEnd.Visibility"
    Value="Visible" />
    <Setter Target="txtStart.Visibility"
    Value="Visible" />
   </VisualState.Setters>
  </VisualState>
 </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
<Grid>
 <Grid.RowDefinitions>
  <RowDefinition Height="6*"/>
   <RowDefinition Height="*"/>
 </Grid.RowDefinitions>
 <Image Grid.Row="0" Source="Assets/
 Singer.jpg"  x:Name="albumArt"
 Stretch="UniformToFill" Margin="0"
 HorizontalAlignment="Center" />
 <Grid x:Name="grid" Grid.Row="1"
 Background="#f0f1f2" >
  <Grid.RenderTransform>
   <CompositeTransform/>
 </Grid.RenderTransform>
 <Grid.ColumnDefinitions>
  <ColumnDefinition Width="*"/>
  <ColumnDefinition Width="*"/>
  <ColumnDefinition Width="*"/>
  <ColumnDefinition Width="*"/>
  <ColumnDefinition Width="Auto"/>
  <ColumnDefinition Width="Auto"/>
  <ColumnDefinition Width="Auto"/>
  <ColumnDefinition Width="*"/>
 </Grid.ColumnDefinitions>
```

```
 <Image Grid.Column="0"  Source="Assets/
 Repeat.png" x:Name="btnRepeat"
 Height="25" Stretch="Uniform" />
 <Image Grid.Column="1" Source="Assets/
 Previous.png" x:Name="btnPrevious"
 Height="25" Stretch="Uniform" />
 <Image  Grid.Column="2" Source="Assets/
 Play.png" x:Name="btnPlay"
 Stretch="Uniform" Height="50" />
 <Image   Grid.Column="3"
 Source="Assets/Next.png"
 x:Name="btnNext"  Stretch="Uniform"
 Height="25"/>
 <TextBlock Margin="20,0"
 VerticalAlignment="Center"  Grid.
 Column="4" Foreground="#a2a7a9"
 x:Name="txtStart" Text="00:13"
 FontSize="30" />
 <Slider  Grid.Column="5"
 Value="50" x:Name="sldProgress"
 VerticalAlignment="Center" Width="450"
 />
 <TextBlock Margin="20,0"
 HorizontalAlignment="Right"
 VerticalAlignment="Center"  Grid.
 Column="6"  Foreground="#a2a7a9"
 x:Name="txtEnd" Text="00:13"
 FontSize="30"  />
 <Image Grid.Column="7" Source="Assets/
 Shuffle.png" x:Name="btnShuffle"
 Stretch="Uniform" Height="25"/>
 </Grid>
</Grid>
```

In this code, we have added the following

1. A Grid with the following controls:
a. 'Image' Controls to display Album Art and other Media Control Buttons (Images are available in the sample code)
b. 'TextBlock' controls to display start and end time
c. 'Slider' control to display the progress

2. We also added two Visual States called *wideView* and *narrowView* to either display or not display the three controls (txtStart, txtEnd, sldProgress) using *VisualState.Setters* property.

3. We then use *AdaptiveTrigger* and *MinWindowWidth* property to enable or disable the state.

Step 3: Press F5 and adjust the width of the app to see how the UI changes. Here are some screenshots of the app on different devices.

Tablet

Windows Phone

Desktop

Here's a link to the gif in case you want to see the app in action.

In future articles, we will add more controls and functionality to this app and make it a fully functional Music Player that looks and behaves different in different devices.

Windows App Studio

As a side note, I thought of mentioning about the Windows App Studio, a free web-based tool that lets anyone create an app with ease. The app studio can now generate apps for Windows 10, as well as for Windows 8.1 and Windows Phone 8.1. Give it a shot!

Conclusion

The Universal Windows Platform Apps aims at changing the way we create apps. Instead of creating separate apps for different platforms, UWP apps would run across all major platforms with minor code changes. With Windows 10, Microsoft aims at bringing different platforms - PC, tablets, phones, XBox, Internet of Things (IoT) together and we sincerely hope that this platform and the benefits it brings, will create an attractive ecosystem for Windows 10 devices ■

Download the entire source code from GitHub at
bit.ly/dncm19-windows10apps

• • • • • • • •

About the Authors

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

Service Oriented Solutions using MSMQ and WCF

There are many types of messaging systems and architectures available at our disposal. All encompass one main objective, which is to be able to send a message (or "object") from a source to its destination, wherever the destination may be.

Some systems commonly send a request to a service (such as WCF services or even communicate with the legacy ASMX web services) for it to do some internal processing (optionally) and finally insert data into the database, such as Microsoft SQL Server. Such systems are very common and are used almost everywhere in today's world of technology.

These systems, like with anything, can have a downtime such as server issues or DB issues and therefore the request coming from the client can fail and therefore no data will be submitted. Sure, you can implement a cluster of servers to attain a higher uptime level, but sometimes this is quite expensive in terms of cost and resources; as you would be dealing with different pieces of software to make it work together. Moreover to make it work in such an environment (SQL Server, Web servers etc...), there will be a number of moving parts to be configured and possibly even re-written/reviewed.

At times, you take a step back and help but wonder if there is any way to simply send a message down the wire without having the systems always being ON all the time, and also without it having to lose messages or data. There must be some way of receiving the message successfully from a client and then be able to process it at a later time, or when other systems are up and running.

This is where MSMQ (Microsoft Message Queuing) comes into play. MSMQ has been in Windows for a very long time and is also included in some Windows CE versions. It allows remote computers to connect and send messages to the destination in a transactional and non-transactional manner and allows you to persist these incoming messages to the disk. So in case of a power failure, the messages are still on the disk, waiting to be processed the next time the system is functional.

You do not always have to have the server with MSMQ up and running. The sender will keep trying to connect and send messages in the queue until the destination it is aiming for, is back up and running; so the client will hold onto the message until there is a connection. This means that the message will not be lost but also that it will automatically be sent when the connection is established.

This is one of the fundamental things about MSMQ that has been with us for years and the best part is that it is available for Free, as it is built into the OS (but you do need to manually install it). It has been available since Windows 95 clients and Windows 2000 for servers with different versions of MSMQ being evolved overtime. But from Windows XP (Professional), it has the ability to communicate via SOAP formatted messages and to be able to multicast messages.

When WCF was developed by Microsoft on the .NET Framework, MSMQ was a story that was included where it would make it easy for developers to integrate with the system. It also meant that WCF and MSMQ can be used to provide secure and reliable transportation of messages which was a big bonus for WCF developers. So with this, you can send and also receive messages to and from the MSMQ queue and also have the ability to read the messages from the queue and start processing them.

MSMQ Usage and some examples

So now that we know what MSMQ is and that we can use WCF on the .NET Framework to queue messages and read them from the queue, where do we use MSMQ? Well the fact is that you can use it anywhere. Some systems use it to communicate with one another if there is no way to interact via services for instance.

Let's see an example of a fictitious goods and a shipping company.

Firstly, the systems are setup to be transactional, so it is guaranteed that if there are multiple operations on different queues for a message, then either all or none of the operations are completed and that nothing is in a "half limbo" state.

Imagine there's a goods company, "DNC Goods", selling goods to customers. Also imagine that the shipping company "DNC Acme Shippers" is the shipping company DNC Goods uses to request and ship orders. Both have MSMQ and Distributed Transactions installed in their system.

DNC Acme Shippers operates between 9-5pm and they do not accept any orders beyond this time. They also do not want to poll a database for orders which have been approved and are waiting to be shipped, because it's expensive to do so and error prone at times. They however need to agree on a format/contract that both systems can understand. This is where WCF is a great candidate to use in such a system as it is based on a contract binding/agreement basis.

DNC Acme Shippers always switch on their systems between these times to receive requests from external companies such as DNC Goods for shipping goods. They don't want to know anything else outside of these hours.

Now imagine a customer logging onto the DNC Goods website, placing an order and paying for the order. The website takes the order request (checks availability etc...) and when the card payment

is approved almost immediately, it will create a message object which will contain the order details and finally send this off to DNC Acme Shippers.

DNC Goods website does not need to know the technical implementation detail such as how the communication channel works or even if it is between the business hours that DNC Acme Shippers are operating on. All the website needs to do is place the MSMQ message on the queue – that is literally it.

MSMQ will then try to connect to the destination (which is provided in the MSMQ message object) and send the message. If it cannot connect, it will try again later. MSMQ takes care of this communication for us, amongst other things that is exposed in the MSMQ message object.

This system provides us the solution for the requirements needed for DNC Acme Shippers where they only want orders for shipping between their business hours and that the message they expect is of a certain agreed standard.

The DNC Goods website processes the order and by executing its own internal logic, processes the card transaction. Finally once the transaction is approved, it creates a message ready to be submitted to DNC Acme Shippers for shipping. The shipping company will take care of the order and email the customer when the goods are shipped.

Setup MSMQ on your System

To setup the system, we will not get into the technical details about setting up a full network architecture, but instead focus more on the software configuration and assume that it is a simple direct end to end connection.
The technologies used for this specific example are as follows:

- Windows 7 Ultimate
- MSMQ (installed from control panel)
- Visual Studio 2012 (non express SKU)
- WCF
- .NET Framework 4.0
- C#

We install MSMQ (Message Queuing) via "Programs and Features/Turn on or off Windows Features" and install with HTTP Support and Multicasting support. We also install Distributed Transactions.

This is all that is needed for DNC Goods.

For DNC Acme Shippers, we need to create a queue, which is transactional. To do so, open the MMC console, and go to File > Add/Remove Snap-ins and select "Computer Management" from the available snap-ins:

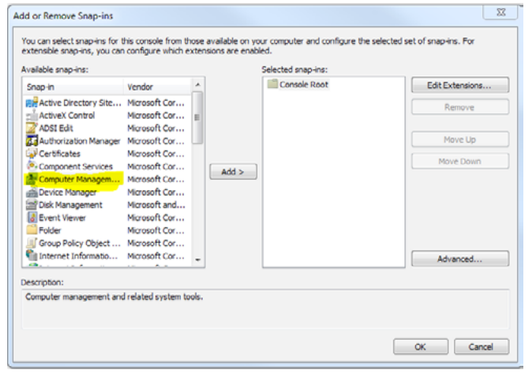


Figure 1. Setting up MSMQ

Press "Add" and then on the dialog screen that appears, press "OK" to connect to the local computer.

Expand the "Services and Applications" and then "Message Queuing". You should see the following:

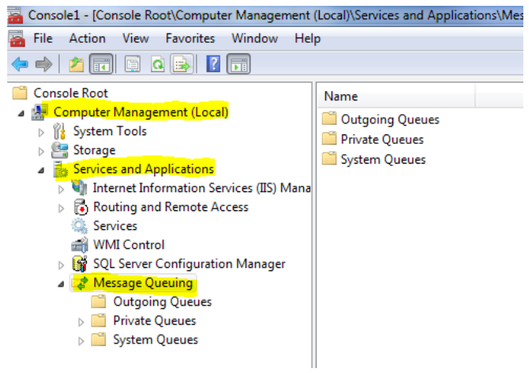


Figure 2. Message Queueing

In the Private Queues, we will create a transaction queue called "ShippingOrders". Right click on the Private Queues and select "New > Private Queue". Type "ShippingOrders" in the Queue name and check the "Transactional" box and press OK:

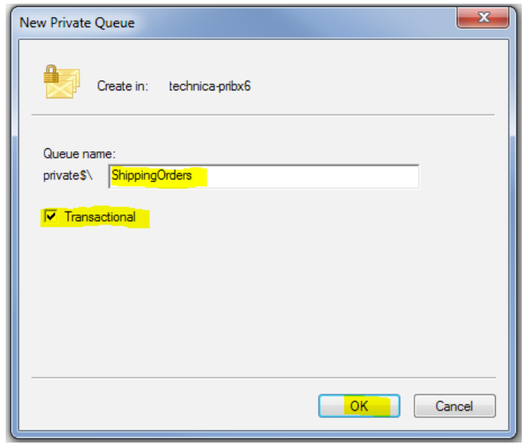


Figure 3. Creating a transactional MSMQ

Now that we have MSMQ installed with a transactional queue created, we need to write some client code which will place an order to the queue. We first need to create a common project which has our contract agreement class "Order" which will be created and placed on the queue.

Create a new project in Visual Studio (using .NET 3.5 or higher) named "Acme.Shared.Contracts" and create a class named Order. Add a reference to "System.Runtime.Serialization" which is a .NET assembly and will expose the attributes we need to apply to the class and properties, in order to serialize the object to and from the MSMQ using WCF as the communication protocol.

The following code will be written as a data contract:

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

namespace Acme.Shared.Contracts
{
  [DataContract]
  public class Order
  {
    public Order()
    {
    }
    [DataMember(IsRequired = true)]
    public int OrderID { get; set; }

    [DataMember(IsRequired = true)]
    public DateTime SubmittedOn { get;
    set; }

    [DataMember(IsRequired = true)]
    public string ShipToAddress { get;
    set; }

    [DataMember(IsRequired = true)]
    public string ShipToCity { get; set;
    }

    [DataMember(IsRequired = true)]
    public string ShipToCountry { get;
    set; }

    [DataMember(IsRequired = true)]
    public string ShipToZipCode { get;
    set; }

  }
}
```

In order to simulate the messages being sent (and received), we first create a simple application (Console/Winforms/WPF) which will create a message and place it on the MSMQ named "ShippingOrders". Be aware that this is not production quality code but simply to illustrate how messages can be sent and received.

Create a new project named "Acme.Dispatcher" and add a reference to the "System.Messaging" .NET assembly, which gives us access to the MSMQ classes to interact with, and the "System. Transactions" assembly which gives us access to the TransactionScope class. Also add a project reference to the "Acme.Shared.Contracts" project, as it contains the data contract we will be using to create and send the Order message. The order message will be sent to the transactional private queue we created above "ShippingOrders".

The code shown here is what is used to create a sample Order and place it on the queue:

```
namespace DNCDispatcher
{
 class Program
 {
  static void Main(string[] args)
```

```
{
 // create a fake order, for
 simulation:
 var anOrder = new Order { OrderID =
 1, ShipToAddress = "123 Abc avenue",
 ShipToCity = "DNC", ShipToCountry =
 "A country", ShipToZipCode = "12345",
 SubmittedOn = DateTime.UtcNow };

 // create a MessageQueue to tell MSMQ
 where to send the message and how to
 connect to it
 var queue = new
 MessageQueue(ConfigurationManager.
 AppSettings["MessageQueuePath"]);

 // Create a Message and set the body
 to the order object above
 var msg = new Message { Body =
 anOrder };

 // Create a transaction
 using (var ts = new
 TransactionScope
 (TransactionScopeOption.Required))
 {
  queue.Send(msg,
  MessageQueueTransactionType.
  Automatic); // send the message
  ts.Complete(); // complete the
  transaction
 }

   Console.WriteLine("Message Sent");
   Console.ReadLine();
  }
 }
}
```

The "MessageQueuePath" is a setting pulled from the app.config file which contains the queue path for MSMQ to know where to send the message to. The app.config setting looks like as follows:

```
<appSettings>
  <add key="MessageQueuePath"
  value="FormatName:Direct=TCP:xx.xx.xx.
  xx\private$\ShippingOrders"/>
</appSettings>
```

The "FormatName" is used when sending messages directly to a computer or over the internet, or reading them while operating in a domain or workgroup environment, or even in an offline mode. It is also used to send messages when authentication, routing and encryption is not needed and in this setup, it is not needed. For more information about *FormatName*, please visit the MSDN resource here: https://msdn.microsoft.com/en-us/library/ms700996(v=vs.85).aspx

We are specifying that we want to send the message to the remote computer. The "xx.xx.xx.xx" should be replaced with the IP Address you are intending on sending the message to.

When you run the simulator, you will see that the message was sent, and when you pull up MSMQ in Microsoft Management Console (MMC), you will see that the number of messages will be set to "1":

Figure 4. Message waiting

Now we know our order is waiting to be read, we can now create a final project which will be the one to read the message from MSMQ. To do this, for this exercise, let us create another Console/Winforms/WPF project named "Acme.OrderReader" and add the "Acme.Shared.Contracts" project reference. Also add the "System.Transactions", "System.Messaging", "System.Runtime.Serialization" and "System.ServiceModel" .NET assembly references for us to be able to process the incoming messages from MSMQ through WCF.

In order for MSMQ to dispatch the message to the reader application through WCF (since WCF is the one to read the messages from the queue), WCF needs to know the type of the object we are expecting so that it can deserialize it and finally dispatch it to our application for processing. To do so, we use the ServiceKnownType attribute.

Let us create an interface called "IOrderInboundMessageHandlerService" and provide a single method that the implementer must implement:

```
namespace Acme.OrderReader.Interfaces
{
  [ServiceContract]
  [ServiceKnownType(typeof(Order))]
  public interface
  IOrderInboundMessageHandlerService
  {
    [OperationContract(IsOneWay = true,
Action = "*")]
    void ProcessIncomingMessage
    (MsmqMessage<Order>
    incomingOrderMessage);
  }
}
```

Following this interface, let's create a class called "OrderInboundMessageHandlerService" which implements this interface and set the ConcurrencyMode enum to "Multiple" so it can process multiple incoming messages. You also have the option of creating a single context service or multiple. This is a design choice that you must decide upon depending on the environment. For this purpose, we will make it "Multiple". As for the transaction behaviour, we will autocomplete the transaction and ensure that a transaction scope is required, since the queue is transactional. With these configuration parameters in mind, the class will look similar to the following:

```
namespace Acme.OrderReader
{
 [ServiceBehavior(ConcurrencyMode =
 ConcurrencyMode.Single,
 InstanceContextMode
 =InstanceContextMode.Single,
 ReleaseServiceInstanceOnTransactionComplete
= false)]

public class
OrderInboundMessageHandlerService :
IOrderInboundMessageHandlerService
{
 #region
 IOrderInboundMessageHandlerService
 Members

 [OperationBehavior
 (TransactionScopeRequired = true,
 TransactionAutoComplete = true)]
 public void
 ProcessIncomingMessage
 (MsmqMessage<Order>
 incomingOrderMessage)
 {
  var orderRequest =
  incomingOrderMessage.Body;
  Console.WriteLine(orderRequest.OrderID);
  Console.WriteLine(orderRequest.
  ShipToAddress);
  Console.WriteLine(orderRequest.
  ShipToCity);
  Console.WriteLine(orderRequest.
  ShipToCountry);
   Console.WriteLine(orderRequest.
   ShipToZipCode);
   Console.WriteLine(orderRequest.
   SubmittedOn);
  }
  #endregion
 }
}
```

As you can see, with the parameters we defined above, we should now be able to receive the message and for the purpose of this exercise, display the details about the order on the console. In reality, you would process the message to the business requirements definition (i.e check the order does not already exist in the system or insert into the database or print out shipping labels etc...).

We now need to host the reader so the reader can read and process incoming messages placed on the queue. To do this, we use the WCF ServiceHost object and then host the OrderInboundMessageHandlerService like so:

```
using System;
using System.ServiceModel;

namespace Acme.OrderReader
{
 class Program
 {
  static void Main(string[] args)
  {
   ServiceHost host = new
   ServiceHost(typeof(
   OrderInboundMessageHandlerService));
   host.Faulted += host_Faulted;
   host.Open();
   Console.WriteLine
   ("The service is ready");
   Console.WriteLine
   ("Press <ENTER> to terminate the
   service");
   Console.ReadLine();
   if (host != null)
   {
    if (host.State ==
    CommunicationState.Faulted)
    {
     host.Abort();
    }
    host.Close();
   }
  }

  static void host_Faulted(object
```

```
  sender, EventArgs e)
  {
   Console.WriteLine("Faulted!"); //
   Change to something more sensible
   – this is just an example showing
what happens when the host has
faulted.
  }
 }
}
```

We are getting very close to completing our project. The final step is to setup the application so that it will open the WCF service host and start reading the messages from the MSMQ. To do so, we use the ServiceHost class. But first, we must configure WCF settings in the config file (app.config). We simply need to tell WCF the details about the service such as A B C (Address, Binding, Contract). The following is what we would enter in the config file:

```
<system.serviceModel>
 <behaviors>
  <endpointBehaviors>
   <behavior name=
   "IncludeExceptionDetails">
    <callbackDebug
includeExceptionDetailInFaults=
    "true" />
   </behavior>
  </endpointBehaviors>
 </behaviors>

 <services>
  <service name="Acme.OrderReader.
  OrderInboundMessageHandlerService">
   <endpoint address="msmq.
   formatname:DIRECT=OS:
   .\private$\ShippingOrders"
   binding="msmqIntegrationBinding"
   bindingConfiguration=
   "IncomingMessageHandlerBinding"
   contract="Acme.OrderReader.Interfaces.
   IOrderInboundMessageHandlerService">
   </endpoint>
  </service>
 </services>
 <bindings>
  <msmqIntegrationBinding>
   <binding
    name="IncomingMessageHandlerBinding"
    closeTimeout="00:30:00"
    receiveTimeout="01:00:00"
    retryCycleDelay="00:00:10"
    receiveRetryCount="0"
    exactlyOnce="true"
    maxRetryCycles="3"
    receiveErrorHandling="Move">
     <security mode="None"/>
    </binding>
   </msmqIntegrationBinding>
     </bindings>
   </system.serviceModel>
```

Notice that the endpoint address is set to look at the local computer MSMQ – this should once again, be changed to the machine where the queue is located on the receiving end. For this exercise, we are sending to a remote computer and the OrderReader is running directly on the machine where the messages are being sent to, thus the reason to read the messages from the local MSMQ endpoint.

We are now finally ready to run the solution. First, fire up the OrderReader app and then secondly fire up the Dispatcher app. The dispatcher will send the message and the OrderReader will almost immediately read the incoming message and display the results in the console:

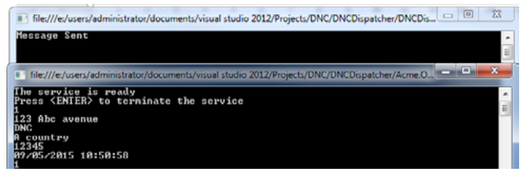


Figure 5. Order sent and received

As you can see, the message was sent and the order was received.

Whilst this solution works – in the real world, things differ slightly. You can have messages that are invalid and the application not expecting it, therefore it would be known as a poison message and MSMQ will automatically place it in its own queue because the reader is unable to process the message (i.e it cannot deserialize it and does not know its type) and terminate the transaction.

You can allow the receiver (OrderReader) to handle these poison messages if you wish within the code. For more information, please visit the following MSDN resource. https://msdn.microsoft.com/en-us/library/aa395218(v=vs.110).aspx

Conclusion

DNC Goods and DNC Acme Shippers now have a solution that both parties are satisfied with, where orders are sent and are only received by DNC Acme Shippers when they open for business during their own business hours. The solution shows us that the technologies are readily available at very little cost, generally speaking. We can create a reasonably straight forward service oriented solution that meets the demands of businesses, using these existing and ever evolving technologies ■

Download the entire source code from GitHub at
bit.ly/dncm19-wcfmsmq

About the Author

After leaving Microsoft, Ahmed Ilyas ventured into setting up a consultancy company, Sandler Ltd, offering the best possible solutions for a magnitude of industries and providing real world answers to those problems. He uses the Microsoft stack to build these solutions and has been able to bring in best practices, patterns and software to his client base. This resulted in him being awarded the MVP title thrice in C# by Microsoft for "providing excellence and independent real world solutions to problems that developers face". His reputation and background has resulted in him having a large client base in the UK and Sandler Software (USA) now includes clients from different industries from medical to digital media and beyond.

What's new in TYPESCRIPT 1.4 AND 1.5 BETA

As JavaScript's popularity continues to grow and a lot of exciting new features are added to its next versions, compile-to-JavaScript languages have a lot of challenges to face. TypeScript, the typed superset of JavaScript from Microsoft has taken up these challenges and continues adding more features to the language to make developers more productive. At the time of this writing, though the language supports comparatively lesser number of features of the next version of JavaScript, the team has promised that it will have full support for ES6 by the time TypeScript 2.0 is released. In addition, popularity of the language got a big push after the **AngularJS team chose to use TypeScript as their language of development** over AtScript.

One of the key reasons why TypeScript started gaining a lot of popularity, is it stays true to ECMAScript standards. This focus on JavaScript makes the language look as close to JavaScript as possible and makes developers still appreciate the syntax of JavaScript. In the recent versions of TypeScript (1.4 and 1.5 beta), the language got some new features that loosens the type system to make it appear a bit more dynamic and even adds some new features of EcmaScript (ES) 6 & 7. We will explore these features in this article.

Using TypeScript 1.5 Beta

The latest version of Visual Studio 2015 CTP installs TypeScript 1.4 on the machine. Installer for TypeScript 1.5 is not yet available as of this writing. It is available through NPM. So you can install it globally using the following NPM command:

- npm install –g typescript

If you have already installed an older version of TypeScript using NPM, you can update it using the following command:

- npm update –g typescript

Union Types

Because of the optional type system in TypeScript, it was not so easy to define a function that takes different types of parameters across different invocations. It was also a challenge to write type definitions for such functions, as we needed to write a declaration for each possible invocation. TypeScript adds a new feature in 1.4 called *Union Types*. This feature allows us to specify multiple types for the same argument without having to write multiple declarations for the same function.

Let us say I have to write a function that accepts either a number, or an array of numbers and returns square of the number, or array of squares depending on the input passed in. Following is the function in TypeScript:

```
function square(nums: number |
number[]): number | number[]{
  if(typeof nums === 'number'){
    return nums * nums;
  }
  else{
    var squares = [];
    nums.forEach(num => {
      squares.push(num*num);
    });
    return squares;
  }
}
```

See the type of argument mentioned in the function. It says, the input can be either a number or, an array of numbers. Similarly, the function returns two types of values, so the return type is also specified in a similar way.

This feature eases the job of writing type declarations for existing libraries. For example, jQuery's `find()` method can be called by passing a string selector, an element or a jQuery object to it. If you see the type declaration for this method in the Definitely Typed project (http://bit.ly/1RImmxS), it has three declarations. They are:

```
find(selector: string): JQuery;
find(element: Element): JQuery;
find(obj: JQuery): JQuery;
```

Using the Union types, it can be replaced with a single declaration, as shown here:

```
find(seo: string | Element | JQuery):
JQuery;
```

Type Aliases

While writing considerably larger apps using TypeScript, it is natural to have the application divided into several modules and each module depending on other modules to achieve its functionality. If a class or an interface defined in a module has to be used a number of times in the importing module, we will have to keep referring to the types using module's reference again and again. Also some types imported from another module may have longer names, or you may want to refer to these types with different names. Thankfully, TypeScript 1.4 adds a feature called *Type Aliases*, which allows us to create alias names for a type.

Say I have a module with some interfaces representing contracts to define different types of cars. To keep it simple, I included two types of cars here, cheap and costly. Following is the module:

```
module Cars{
  interface ICar{
    manufacturer: string;
    price: number;

    drive(speed: number);
  }

  export interface ICheapCar extends
  ICar
  {
    mileage: number;
  }

  export interface ICostlyCar extends
  ICar
  {
    length: number;
    width: number;
  }
}
```

The only way we know to refer to these interfaces, is by using the module name.

```
var car1 : Cars.ICheapCar;
var car2 : Cars.ICostlyCar;
```

Using this syntax over and over to refer to these types would tire us. So, let's create aliases. Following snippet creates the aliases:

```
type IEconomicCar = Cars.ICheapCar;
type ILuxuriousCar = Cars.ICostlyCar;
```

Now you can use the alias names to refer to the interfaces.

```
var car1 : IEconomicCar;
var car2 : ILuxuriousCar;
```

It is also possible to create types on primitive types. Here are some examples:

```
type setOfChars = string;
type numericValues = number;
```

We can create aliases on mixed-generic types as well. Here's an example:

```
type CarCollection = Array<IEconomicCar
| ILuxuriousCar>;
```

The type *CarCollection* can be used to store a list of objects of both *IEconomicCar* and *ILuxuriousCar* types.

The *typeof* and *instanceof* operators can be applied on variables defined using type aliases to check their type before performing an operation. For example, the following snippet defines two classes and a collection to hold objects of these types:

```
class Farmer{
  startFarming(){
    console.log("Started. Don't disturb
    me for 3 hours from now.");
  }
}

class Carpenter{
  buildADoor(){
    console.log("Will start today. Meet
    me after 10 days.");
  }
}

type WorkersCollectionType =
Array<Farmer | Carpenter>;
var workers: WorkersCollectionType = [];
workers.push(new Carpenter());
workers.push(new Farmer());
for(var count = 0; count<workers.length;
count ++){
  if(workers[count] instanceof
  Carpenter){
    workers[count].buildADoor();
  }
  else{
    workers[count].startFarming();
  }
}
```

The loop that iterated over the items in the collection checks for the type of the instance, before it performs an operation.

Better Generics and Generic Type Inference

Because of *alias* types, arrays infer types from the value assigned to them during declaration. For example, consider the following snippet:

```
var arr = [10, new Carpenter()];
arr.push(161);
arr.push("Ravi");  //not allowed
arr.push(new Carpenter());
```

As the array is initialized with a number and an object of *Carpenter* type, type of the variable is assigned as an array of a union type. So the first statement is similar to:

```
var arr2:Array<number | Carpenter> =
[10, new Carpenter()]
```

As string is not compatible with any of these types, an attempt to insert a string value into the array results in an error.

Generics have been made stricter and they restrict assigning values of incompatible types on two assignment targets declared using the same generic notation. Following is an example of strict generics:

```
function add<T>(first: T, second: T) : T{
  if(typeof first === 'number' || typeof
  first === 'string') {
    return first + second;}
  return null;
}
console.log(add(1, 25)); //26
console.log(add("firstName","lastName"));
//firstNamelastName
console.log(add([1,2,3],[3,4,5])); //
null
console.log(add(1, "Ravi")); //Error
```


Const Enums

We use enums to store a list of fixed values and these values are collectively used to represent a set of values. On compilation, TypeScript generates an object for the enum and the values are assigned to properties in the object.

When marked with the keyword *const*, the compiler doesn't create an object for the enum. So it is not allowed to access the const enum as an object in TypeScript. We can use the values alone and the compiler replaces all usage occurrences with their corresponding values.

```
const enum Days {Sunday, Monday,
Tuesday, Wednesday, Thursday, Friday,
Saturday};
var days = Days; //not allowed
console.log(Days.Monday); //0
```

ES6 Features

As I stated in the introduction of my first article on ES6 (http://www.dotnetcurry.com/javascript/1090/ecmascript6-es6-new-features), ES6 got a lot of features from some compile-to-JavaScript languages and from popular libraries. ES6 got some of its features like classes, arrow functions, some part of module system and a couple of others from TypeScript as well. So some of the ES6 features are already available in TypeScript. The team started implementing features of ES6 into the language and TypeScript 1.5 is going to have a decent support for ES6.

ES6 output mode

By default, TypeScript code gets converted to ES5 or, ES3 version of JavaScript. Now we can transpile TypeScript code to ES6 code using a compilation option. Use the following command:

- tsc --target ES6 file.ts

Let and Const

The JavaScript we use today doesn't have block level scoping. Any variable declared using the *var* keyword at any level in a function, is hoisted at the beginning of the function. ES6 adds block level scoping by introducing a new keyword, *let*. TypeScript has got support for this keyword now. The *let* keyword can be used to declare intermediate variables that store temporary values, like counter in *for* loop. Here is an example:

```
for(let c = 0; c< 10; c++){
  console.log(c*c);
}
```

The *const* keyword in ES6 is used to define scoped constants. TypeScript now supports this keyword. Any attempt to reassign value of a constant would result in an error. The following function uses the *const* keyword:

```
function findArea(radius: number):
number{
  const pi=3.4159;
  return pi * radius * radius;
}
console.log(findArea(20));
```

Template Strings

Appending values to strings in JavaScript has never been enjoyable. ES6 adds support for template strings to make it easier to add values of variables to a string and to easily assign multi-line values to strings. TypeScript 1.4 adds this feature to the language. The following snippet uses this feature:

```
var person={
  firstName:'Ravi',
  lastName:'Kiran',
  occupation:'Author'
};
console.log(`${person.firstName}
${person.lastName} is a/an ${person.
occupation}.`);
var template=`<div>
  <span>Some text goes here...</span>
</div>`;
```

Destructuring

Destructuring is a feature added to JavaScript in

ES6 that saves a lot of time in extracting values out of arrays and objects. It defines a shorter way of assigning the values from arrays and objects into variables. TypeScript 1.5 adds the support of destructuring to the language and it gets transpiled to its best possible alternative in the target version.

Following are a couple of examples of destructuring:

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

var [first, second, third ] = numbers;
//first: 20, third: 40

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

var {name, browserStatus:{opera}} =
topic;  //name: 'ECMAScript 6', opera:
'partial'
```

Modules

Support of *modules* is one of the most important features added to ES6. Addition of modules makes it easier to structure the code and manage the dependencies easily without need of an external library. As the feature is not yet implemented by browsers, we need to rely on an existing module system like AMD or CommonJS today to manage JavaScript dependencies in production environments. Though TypeScript has its own module system, it now embraces the module system of ECMAScript 6 and provides a way to transpile the ES6 modules into either AMD or, CommonJS system.

If you are not already familiar with the syntax and usage of ES6 modules, check the article on ES6 modules on DotnetCurry (http://www.dotnetcurry.com/javascript/1118/modules-in-ecmascript6).

Consider the following code. It is a piece of TypeScript code using the export statement of ES6 to export objects out of the module.

```
class Employee{
  id: number;
  name: string;
  dob:Date;

  constructor(id, name, dob){
    this.id = id;
    this.name=name;
    this.dob= dob;
  }

  getAge(){
    return (new Date()).getFullYear() -
    this.dob.getFullYear();
  }
}
var [x, y] = [10, 20];

function getEmployee(id, name, dob){
  return new Employee(id, name, dob);
}

export {Employee, getEmployee};
```

Say, the file is saved as employee.ts. Following command would transpile the module into a CommonJS module:

```
tsc employee.ts --module commonjs
--target ES5
```

Now you can run the browserify command over the generated file and load the file in a browser. To use browserify, you need to install the NPM package of browserify globally. The following command does this for us:

```
npm install -g browserify
```

Here is the browserify command to create a bundle file containing the above file:

```
browserify employee.js > bundle.js
```

The bundle file is a self-contained file and it can be loaded into the browser without importing any other external scripts.
You may check the official site of browserify (http://browserify.org/)if you want to learn more.

As you can see, the command accepts the *module* flag in addition to the *target* flag discussed earlier. Using this flag, the TypeScript file can be transpiled into either a CommonJS module or, an AMD module.

Decorators (ES7)

Decorators are not added to the specification of ES6; instead they are a part of the ES7 spec. Using decorators, a JavaScript object can be extended declaratively. This feature is already used in Angular 2.0 and in Aurelia (http://aurelia.io/). These frameworks use this feature extensively to achieve things like Dependency Injection, making a class a component, to make a field bindable, and many more.

Though the name sounds like extra burden, defining and using a decorator is fairly simple. A decorator is a function that accepts the object to be decorated, name of the property and object descriptor. It has to be applied on a target using the "at the rate" (@) symbol. Following is the signature of a decorator function:

```
function decoratorFunc(target, name,
descriptor){
  //body of the function
}
```

The following snippet defines and uses a decorator:

```
function nonEnumerable(target, name,
descriptor){
  descriptor.enumerable = false;
  return descriptor;
}

class Person {
  fullName: string;

  @nonEnumerable
  get name() { return this.fullName; }

  set name(val) {
    this.fullName = val;
  }

  get age(){
    return 20;
  }
}
```

```
var p = new Person();
for(let prop in p){
console.log(prop);
}
```

As the property name has been made non enumerable using a decorator, the "for...in" loop prints the property age alone on the console.

Decorator Metadata

Metadata Reflection API is another proposed feature for ES7 (link to proposal - http://bit.ly/1SUthGZ). This API is designed to be used along with the decorators to implement features like Dependency Injection, perform runtime type assertions and mirroring. The feature is already in use in Angular 2 for DI and to declare components.

To use this feature, we need the polyfill of Reflection API. It can be installed using the following NPM command:

```
npm install reflect-metadata
```

We can either import this library into the TypeScript file using the ES6 module syntax or, we may even load this script in the browser before the script using it, loads to make the API available.

Now we can start defining the metadata annotations using this API and start using them. Following is a decorator that uses the metadata API:

```
function Inputs(value: Array<string>) {
  return function (target: Function) {
    Reflect.
defineMetadata("InputsIntoClass", value,
target);
  }
}
```

This is a simple decorator that accepts the Type of data passed into a class or, a function. One can extend this idea to create a dependency injection system.
The following class uses this decorator and passes the metadata:

```
@Inputs(["Employee"])
```

```
class MyClass {

  emp: Employee;
  constructor(e: Employee){
    this.emp = e;
  }
}
```

Now we can get the value of metadata applied on this class using the metadata read APIs. The following snippet reads the metadata of this class:

```
let value: Array<string> = Reflect.
getMetadata("InputsIntoClass", MyClass);
console.log(value);
```

The *console.log* statement in the above snippet prints value of the metadata passed into the decorator, which is an array containing a single value in this case. You can read this information to create an abstraction to instantiate the class.

Conclusion

The TypeScript team is putting together a lot of work to make the language better for larger applications and by keeping it as close to JavaScript as possible. The new features and the support for ES6 helps to keep the language relevant in modern JavaScript world too. The final version of TypeScript 1.5 may include a couple of additonal features and the next version will have support for async/await (part of ES7 spec). We will keep you updated with these features as they release ■

About the Author

ravi kiran

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