ISSUE 23 MAR-APR 2016

DNCMagazine

www.dotnetcurry.com

ROSLYN
An Overview

Metaprogramming in ES6 using Symbols

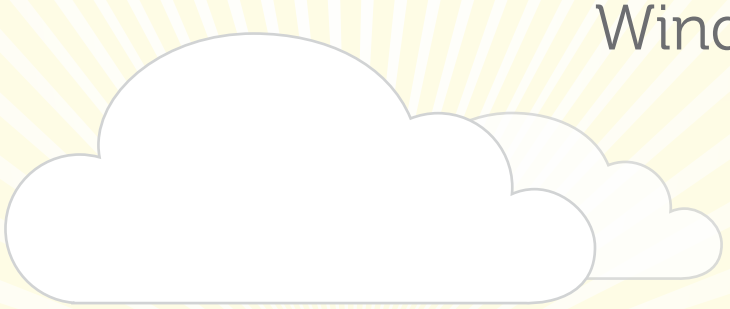
Working with NoSQL using Azure DocumentDB

Azure Active Directory Essentials

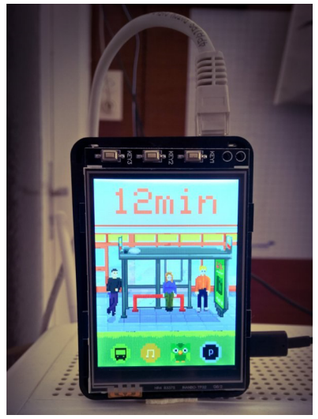
SOLID: Interface Segregation Principle

Liskov Substitution Principle : A perspective

Home Pi Windows 10 IoT



JS ES6



CONTENTS

Editor in Chief

Hello friends! Our 23rd edition warms up with an article by Damir on C# compiler "Roslyn" where he gives you a good overview of the compiler and spikes your interest to explore it further.

Next we have two articles on SOLID principles where first time DNC author Yacoub shares an interesting perpective on the Liskov Substitution Principle, followed by an article from Craig who talks about Interface Segregation Principle in his popular Software Gardening column. For our devs interested in the Cloud platform, we have two good articles by Mahesh and Kunal on Azure DocumentDB and Active Directory Fundamentals for MVC developers.

Shoban follows up on his cool Windows 10 and Raspberry Pi IoT project called HomePi and adds some more features to his Bus schedule app, whereas for our JavaScript developers, Ravi nicely demonstrates how Symbols in ES6 add the ability of meta programming to JavaScript. Enjoy!

Feel free to email me at **suprotimagarwal@dotnetcurry.com**

Suprotim Agarwal

EDITORIAL

POWERED BY

a2Z | Knowledge Visuals

CREDITS

Editor In Chief
Suprotim Agarwal
suprotimagarwal@
a2zknowledgevisuals.com

Art Director
Minal Agarwal
minalagarwal@
a2zknowledgevisuals.com

Contributing Authors
Craig Berntson
Damir Arh
Kunal Chandratre
Mahesh Sabnis
Ravi Kiran
Shoban Kumar
Yacoub Massad

Technical Reviewers
Damir Arh
Suprotim Agarwal

Next Edition
3rd May 2016

www.dotnetcurry.com/magazine

.NET & JavaScript Tools

Shorten your Development time with this wide range of software and tools

CLICK HERE

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®

ROSLYN
An Overview

Roslyn has been known as the code name for the next generation of C# compiler, at least since its first public preview was released in 2011. Infact the project started internally at Microsoft a couple of years earlier. Even before its first final release in Visual Studio 2015, it started to mean a lot more than just a new compiler. At that time, it also got a new official name: **.NET Compiler Platform**. Nevertheless, the word Roslyn is still a part of developer vocabularies and will probably remain in use for quite some time. Let's look at what one might be referring to today when mentioning Roslyn, what the current state of the project is, and how it can benefit developers.

.NET Compiler Platform (a.k.a Roslyn)

Most developers treat compilers as black boxes: they receive source code as input, do some processing on it, and output executable binaries. Computer science graduates, who have taken a compiler course during their studies, might remember that internally compilers have a pipeline structure. Source code processing consists of three main phases: syntax analysis, semantic analysis, and code generation. Although each phase outputs its own intermediate results and passes them as input to the next phase, these are just internal data structures that are not accessible outside the compiler.

In the early days of software development, when source code at its very best was written in simple text editors, and compilers were invoked from the command line, this sufficed. However, throughout the years, our expectations have changed a lot. Modern integrated development environments, such as Visual Studio, offer so much more: syntax highlighting, code completion, debugging, refactoring, static analysis, etc. Even in simple programming text editors, such as Sublime Text, we expect at least syntax highlighting and code completion.

Most of these services require some level of knowledge about source code. Since compilers do not share the intermediate results of their processing, the editors need to repeat parts of the same work. For example, the editor in Visual Studio 2013 and earlier versions, did a lot of C# language processing independently of the C# compiler. Not only that; even their code bases were separate, potentially causing them to behave differently.

Image 1: Compiler as a black box

When Microsoft decided to rewrite the C# and Visual Basic compilers from scratch (because their code base became a challenge to maintain and adding new features became unfeasible), they did not want to improve just the code. They rather wanted to make it useful in other scenarios as well: diagnostics, static analysis, source code transformation, etc. In order to achieve that, Microsoft created a compiler that not only converts source code into binaries, but also acts as a service, providing a public API for understanding the code.

Public API

Roslyn APIs reflect the pipeline architecture of a traditional compiler, giving access to each step of compiler's source code analysis and processing:

- **Syntax tree API** exposes the lexical and syntactic structure of the code, including formatting and comments. The latter two are irrelevant for later phases of the compiler pipeline, but important for tools that want to manipulate the code and keep the formatting and code comments intact.

- **Symbol API** exposes the symbol table containing names declared in the source code, as well as those originating from referenced assemblies without corresponding source code.

- **Binding and Flow Analysis APIs** exposes the complete semantic model of the code that becomes available after the binding phase. These APIs contain all the information about the code that is required for generating the binaries, including any errors and warnings that were detected in the code.

- **Emit API** provides access to services for emitting the IL byte code.

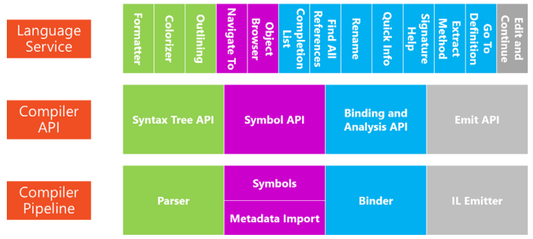


Image 2: Compiler as a service

The scope of Roslyn is not limited to the compiler API. Most of its integration into Visual Studio 2015 is part of the public API as well, except for the thin layer interacting with its proprietary code:

- **Diagnostic API** enables development of custom third party code diagnostics that are pluggable into Visual Studio code editor and MSBuild build system, making them indistinguishable from compiler's built-in diagnostics for detecting errors and warnings in code.

- **Scripting API** provides the interactive C# environment in Visual Studio 2015 Update 1 – the so-called REPL (read-eval-print loop).

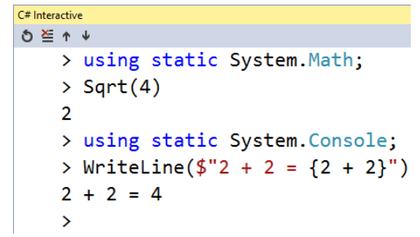


Image 3: C# interactive in Visual Studio 2015 Update 1

- **Workspaces API** provides the model of a complete solution, consisting of multiple projects with documents and assembly references, making it possible to drill down all the way to the semantic and syntactic models of the code.

Roslyn code base is now being used in two scenarios with almost opposite requirements. For the compiler, high throughput and correctness are most important; whereas for the interactive editor, responsiveness and error tolerance are of higher priority. Roslyn managed to achieve comparable, if not better performance than Visual Studio

2013, in both fields. However, this required certain compromises that are evident from the API. All of the data structures (syntax trees, semantic model, and workspaces) are immutable; i.e. they cannot be modified.

Every time a developer changes a single character in any of the files, a new copy of all the data structures is created, leaving the previous version unchanged. This allows a high level of parallelism and concurrency in the Roslyn engine, as well as in its consumers, thereby preventing any race conditions to occur. Of course, in the interest of performance, these operations are highly optimized and reuse as much of the existing data structures as possible. Again, those being immutable makes this possible!

Immutability of data structures affects API consumers in a significant manner:

- None of the models exposed by the API can be modified. Instead, they provide methods for creating copies with a specific change applied to them.

- Since no data structure ever changes, events make no sense. Instead, user code can register to be called when a new version of the model is created, and the models can be compared to detect the differences.

Immutability does require some getting used to; but in the end, it does not have a negative effect on what can be achieved using the API. After all, C# and Visual Basic support in Visual Studio 2015 is completely based on the same public API.

Roslyn in Action

As mentioned above, Visual Studio 2015 is taking full advantage of **.NET Compiler Platform, also known as Roslyn**. Many of the code editor functionalities are implemented using Roslyn public API: from automatic code formatting and coloring; to IntelliSense, code navigation and refactoring. Since the full public API is available as open source, other integrated development environments are already switching from their internal implementations, to using Roslyn as the engine powering the advanced code editor features. At the time of writing, preview versions of Xamarin Studio and Mono Develop are available with .NET compiler platform already integrated.

To bring the power of Roslyn to other text editors that do not run on .NET and cannot host Roslyn in process, OmniSharp project has been brought to life. It is a cross-platform wrapper around Roslyn, exposing its API across HTTP. This makes it accessible to any text editor, and there are already integrations available for many popular programming text editors of today: Atom, Brackets, Sublime Text, Visual Studio Code, and others.

With the help of Roslyn, code diagnostics and refactoring became accessible to individual developers. This resulted in several new specialized extensions for code analysis, the most popular of them being Refactoring Essentials, Code Cracker, and CSharp Essentials. They are free and do a lot of what was once only possible using large expensive commercial extensions like ReSharper, CodeRush, and JustCode. It will be interesting to see, how these extensions will adapt to the changes. DevExpress has already released a preview of CodeRush for Roslyn – a rewrite of the original CodeRush extension. JetBrains has announced that they do not plan to use Roslyn in future versions of ReSharper, while Telerik as of this writing, has not shared any news about their plans for Just Code.

Of course, Roslyn is not limited to being useful only in code editors. Standalone tools that need to process C# or Visual Basic code can benefit as well. The two most prominent open source projects in this field are scriptcs and Source Browser. The former makes C# a scripting language, allowing simple scripts written in C# to be executed directly from command line. The latter is powering Microsoft's reference source web site – the place to browse .NET source code online with full syntax highlighting, navigation and search. Source Browser project can be used to host any other .NET project source code online in a similar manner.

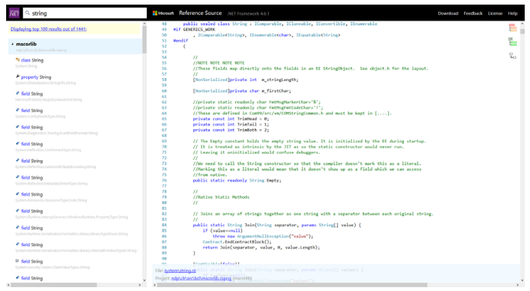


Image 4: .NET Framework Reference Source web site

This is far from an exhaustive list of publicly available projects using Roslyn, let alone proprietary ones being developed for internal use only. Still, they clearly show that the possibilities opened up by public Roslyn APIs are often limited only by imagination.

Getting Engaged with Roslyn

The most obvious way to benefit from the Roslyn open source project is probably by using its APIs to develop custom code diagnostics, refactoring or standalone tools. Two of my articles that can help you get started have already been published in previous editions of the DotNetCurry (DNC) magazine: Diagnostic Analyzers in Visual Studio 2015 and Create Your First Diagnostic Analyzer in Visual Studio 2015. There is also a lot of documentation and samples on the official Roslyn open source site. In spite of that, development with Roslyn is not trivial, and it might not seem very applicable to your daily work.

However, there are other ways to get involved in Roslyn open source community. Having all the source code available allows you to learn from it or check how something is implemented. You could even modify the code and create your own private version of the compiler, for instance. Except for learning purposes, this does not make much sense. It is a much better idea, to post the bug report or feature suggestion as a GitHub issue and get feedback from the team. If they approve it, either you could have your code become a part of the official release, or somebody else might implement the bug fix or feature instead of you.

Not only is all the code openly available, the language is designed out in the open as well. Currently the language team at Microsoft is planning and designing new C# 7 features. They are publishing all their notes as GitHub issues and accepting comments from the community. If you want to see what is the next version of C# bringing in, or maybe even influence how the features work and what syntax they use, NOW is the time to get involved.

Conclusion:

Although Visual Studio 2015 with Roslyn compilers has been released about 8 months ago (July 2015), project Roslyn is continuing, and is more alive than it ever was. The community is taking part along with internal Microsoft teams, submitting feedback and contributing code. Improvements to .NET Compiler Platform are being done constantly and are released as part of Visual Studio updates. Next version of languages are currently in a planning and design phase. Third party developers are learning about Roslyn and finding innovative ways to use it in their own software projects. If you are interested in any of the above, do not hesitate to take a closer look at Roslyn. You just might end up liking what you discover!

About the Author

damir arh

MVP Microsoft® Most Valuable Professional

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

A PERSPECTIVE ON THE LISKOV SUBSTITUTION PRINCIPLE AND THE COMPOSITION ROOT

The Liskov Substitution Principle (LSP) is an object-oriented design principle that puts some restrictions on the classes that inherit other classes or implement some interfaces. It is one of the five SOLID principles that aim to make the code easier to maintain and extend in the future.

In simple terms, LSP says that derived classes should keep promises made by base classes. This also applies to interfaces, and it means that classes that implement some interface, should keep the promises made by that interface.

In more formal terms, this is called a contract. And in order not to violate LSP, classes must make sure that they abide by the contract (that they have with their consumers) which they inherited from the base class or the interface.

One definition of LSP is that "Subtypes must be substitutable for their base types". It is not clear though what "substitutable" means exactly.

Some people say that this means that if some class, say ClassA, depends on IService; then you should be able to inject any class that implements IService into ClassA without breaking the application. And if this is not the case, i.e. one of the implementations of IService would break the application if it was injected into ClassA, then they would consider that this implementation is not actually an "IService" and thus they would create a new interface for it.

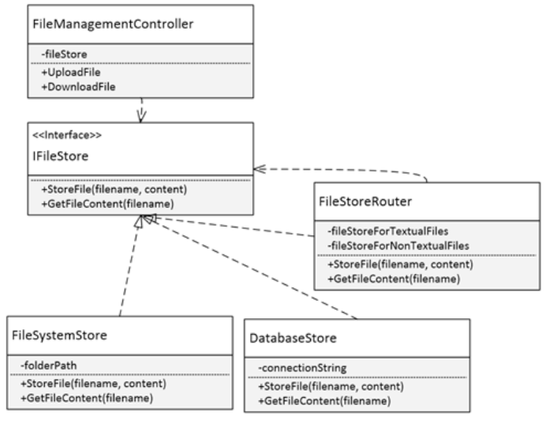
In this article, I am going to argue that this is not the case (or atleast that it shouldn't be the case), and that we should look at LSP in terms of contracts only.

Although the examples I provide in this article are for interfaces, a similar discussion can be made for base classes.

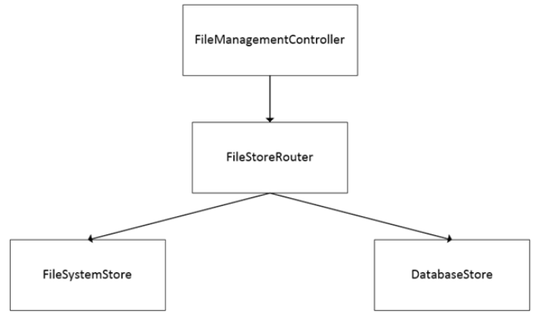
An example

Let's start with an example. Let's say we have a web site that allows users to back up their files by uploading them to the server. It allows them also to download the files later if they want to. Let's assume that the requirements state that textual files should be saved to a database so that their content can be indexed and later queried. Other files should be saved to the file system. Now let's say that we decided to design such system like this:

The following figure is a UML diagram for the involved types:



And the following graph shows the object graph that is composed in the Composition Root:



And here is how it is composed in code (with the C# language):

```
var fileManagementController = new
FileManagementController(
 new FileStoreRouter(
  fileStoreForNonTextualFiles: new
  FileSystemStore(folderPath),
  fileStoreForTextualFiles: new
  DatabaseStore(connectionString)));
```

The FileManagementController class is an ASP.NET controller. It receives requests to upload files via the UploadFile() method. The FileManagementController has a dependency on IFileStore (via constructor injection) and it simply uses such dependency to store the file (by invoking the StoreFile method). The FileManagementController does not care how the file store works. As far as it is concerned, its job ends by giving the file to the IFileStore dependency. In the Composition Root, the FileManagementController is injected with a FileStoreRouter. This class receives two IFileStore implementations in the constructor; *fileStoreForTextualFiles* and *fileStoreForNonTextualFiles*. The responsibility of this class is to decide whether the file is a textual file and to route method invocations to the correct IFileStore.

The FileManagementController class also receives requests to download files that were previously uploaded, this is done through the DownloadFile() method.

In the Composition Root, into FileStoreRouter, an instance of FileSystemStore is injected for fileStoreForNonTextualFiles, and an

instance of DatabaseStore is injected for fileStoreForTextualFiles.

Now, it is clear that if we swap FileSystemStore and DatabaseStore, i.e., we inject FileSystemStore for fileStoreForTextualFiles and DatabaseStore for fileStoreForNonTextualFiles, we break the application because the requirements state that textual files should go to the database, not the file system.

Now, the question is: **do these classes, i.e., FileSystemStore and DatabaseStore violate the Liskov Substitution Principle?** Should we create IFileSystemStore and IDatabaseStore and make FileStoreRouter depend on these two interfaces instead?

I am arguing that the FileSystemStore and DatabaseStore classes *do not* violate LSP.

Contracts

Let's think about the IFileStore contract.

The contract of the IFileStore interface has not been formally defined, so let's try to define it here based on what makes sense.

The implementer of the interface is obliged to:

- Receive the file upon the invocation of StoreFile and store it somewhere, it doesn't matter where.
- Return the file content later if the GetFileContent() method was invoked given a name of a file that was stored previously.
- In case the file does not exist, the GetFileContent() method is expected to throw a FileNotFoundException exception.
- Accept a filename that has a length that is less or equal to 250 characters and that contains alphanumeric characters or the dot character.
- Accept a file content that is of size less or equal to 10MB.

The consumer of the interface is obliged to:

- Not pass a file name whose length exceeds 250 characters or that contains characters that are not alphanumeric and that are not the dot character.
- Not pass null values.

That's it. That is our definition of the IFileStore contract.

An implementation of the IFileStore interface violates the LSP if it violates any of the conditions of the contract. For example, if one implementation does not accept an alphanumeric filename that has a length of 200 characters, then it violates LSP.

Simple and Complex Contracts

Some contracts have more conditions than others in terms of number and complexity. For example, some contract conditions require that we invoke methods in a certain order. This is called temporal coupling. Others might require that the parameters that we pass have certain constraints. The contract for IFileStore has such a condition. Other conditions require that we ask the dependency some question before we invoke some method to see it we can invoke it. An example of such contract is the contract for the ICollection<T> interface in the .NET framework; it contains a property called *IsReadOnly* that we can use to determine if we are allowed to invoke methods that change the content of the collection.

It follows logically that the more conditions a contract has in terms of number and complexity, the easier it is for an implementer of the contract interface to violate LSP. Simply put, there would be more ways in which it could violate LSP.

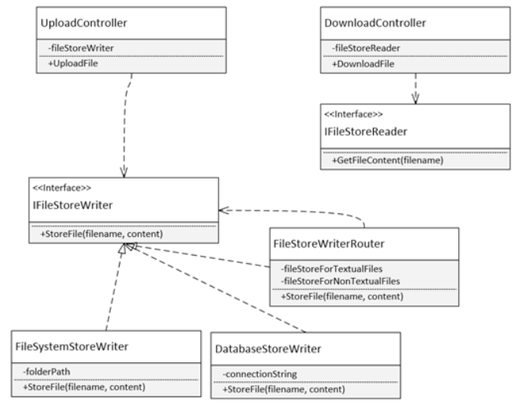
THE SINGLE RESPONSIBILITY PRINCIPLE (SRP) AND THE INTERFACE SEGREGATION PRINCIPLE (ISP)

Two other principles of SOLID design are the SRP and the ISP. The SRP simply states that a class should do one thing only and thus has only one reason to change. ISP states that a client should not be forced to depend on methods that it does not use.

If we apply the ISP, the result would probably be more interfaces that contain lesser methods. And if we apply the SRP, the result would be more classes that contain lesser methods. In summary, applying these two principles will make our contracts simpler (since the interfaces are smaller) and will generate more classes that each abide to a simpler contract. So, applying these principle makes it easier for us not to violate the LSP.

We can apply these principles to the classes and interfaces in the example we just saw. In our example, the ISP is not violated because the FileManagementController depends on both of the methods of IFileStore. However, we could argue that the SRP is violated by having a single controller process both upload and download requests. We can split it into two controllers; UploadController and DownloadController. Now, each one of these controllers require only a single method from the IFileStore interface. Now, they do violate the ISP.

To not violate the ISP, we can split the IFileStore interface into two interfaces: IFileStoreWriter and IFileStoreReader. We also split each class into two; one for reading and one for writing. Here is how the types would look like:



Please note that this UML diagram does not show all of the types that are related to the IFileStoreReader interface for reasons of brevity. These skipped types are the FileStoreReaderRouter, FileSystemStoreReader and DatabaseStoreReader classes.

The original contract stated that the GetFileContent() method should return the content of a file that was previously stored using the StoreFile() method. Now, after we split it into two contracts, no individual class or interface has such responsibility since no class or interface has the two methods together. Where did this responsibility go?

THE COMPOSITION ROOT

Applying these SOLID principles, we are making our individual classes and interfaces simpler. Instead of having few big classes that have big responsibilities and know much about the system, we are having small and simple classes that have smaller responsibilities and that know less about the system. We are having more contracts that each have lesser conditions.

Where is this responsibility going? Where is this knowledge of the system going?

To the Composition Root.

The Composition Root is the place in an application where we wire all our classes together to create the object graph that *constitutes* the application. After we apply the SRP and the ISP, the responsibility of the Composition Root is much higher.

Before applying these principles, the Composition Root had only a few big puzzle pieces to put together. Now, it has a lot of smaller puzzle pieces that it needs to put together.

In our example, the Composition Root is responsible for making sure that the IFileStoreReader implementation that is injected into the DownloadController, can return the content of files uploaded by using the IFileStoreWriter implementation injected into the UploadController.

Before splitting IFileStore into these two interfaces, the Composition Root had no such responsibility. The IFileStore implementation had this responsibility then.

Although it is harder to break the LSP now, we can still break the application by composing our object graph in an incorrect way. E.g., by injecting some implementation of some interface into the wrong place.

But now, the responsibility of not breaking the application is where it should be; the entity that constitutes the application, i.e., the Composition Root.

To summarize the responsibilities:

- Individual classes have the responsibility to abide by the implementer part of the contracts they implement (usually a single contract if we apply the SOLID principles)

- Individual classes have the responsibility to abide by the consumer part of the contracts of the dependencies that they have.

- The Composition Root has the responsibility of creating the individual objects and wiring them correctly so that the application does what it is supposed to do. This is not an easy thing to do because individual class are highly composable and we can easily compose them in a way that does not match the application's requirement.

THE NULL OBJECT PATTERN

One pattern that is related to the argument of this article is the Null Object Pattern. With this pattern, we create an implementation of a specific interface that actually does nothing. There are many uses for such a pattern. For example, sometimes when we want to modify the application to turn off a specific feature. Instead of changing the code in the consuming class, we simply inject a Null Object to it.

For example, in the case of our IFileStoreWriter, we could create a NullFileStoreWriter that does nothing when the StoreFile method is invoked.

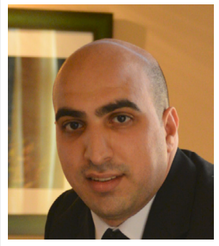
Imagine what would happen if we inject this implementation into every place where IFileStoreWriter is expected. This will most probably break the application. **Does the Null Object Pattern violate LSP?**

SUMMARY:

When we apply SOLID principles, the SRP and the ISP in particular, we get a large number of classes and contracts. Such configuration means that we have a set of highly composable components that we initially compose in a particular way, but that can be easily composed in a different way to meet new requirements.

Such composition happens in the Composition Root. In this article, I argued that the Composition Root has a bigger responsibility of not breaking the application and thus individual classes need only care about not violating the contracts they deal with in order not to violate the LSP.

About the Author



Yacoub Massad is a software developer that works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software. You can view his blog posts at yacoubsoftware.blogspot.com.

Image Source: https://azure.microsoft.com

Working with NoSQL using Azure DocumentDB

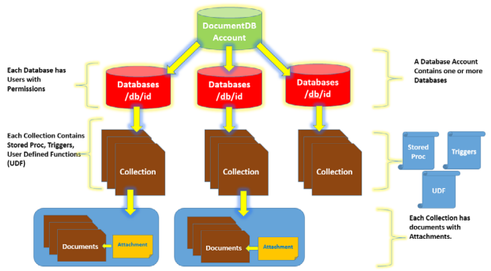
NoSQL has emerged as an alternative to relational databases. NoSQL does not represent a single product or a single technology. It represents a diverse set of concepts about data storage and manipulation. The motivation behind NoSQL is simplicity, scalability, agility and better control over data availability.

Microsoft entered the NoSQL database space via its Azure platform. Azure provides two NoSQL options: **DocumentDB** and **Azure Table Storage**. DocumentDB is a schema free database service fully managed by Microsoft and was designed to natively support JSON documents. It automatically indexes all documents for relational and hierarchical queries. DocumentDB is reliable, highly scalable and can store several terabytes of data. From a learning perspective, if you are familiar with SQL Server or SQL, you can easily pick up DocumentDB.

Important Features of DocumentDB

- Provides default index to all documents in a database.
- Provides support for SQL query like syntax to query JSON documents.
- Uses standard scalar, string functions in a query.
- Provides support for Stored Procedures, User Defined functions, Triggers, etc.

The database entities that DocumentDB manages are referred to as resources, uniquely identified by a logical URI. The Resource Model structure of DocumentDB is shown in the following diagram:



As illustrated in the diagram, each DocumentDB account contains one or more databases. Each database can have one or more users, with permissions associated with them. Each database further contains *document collection*. The collection contains data along with stored procedures, triggers and user defined functions, using which data can be manipulated.

How to Create DocumentDB Database, Collections?

Since we are using Microsoft Azure, we need an Azure subscription to create and work with the DocumentDB. Please visit this link to go through the free trial, pricing details and the documentation.

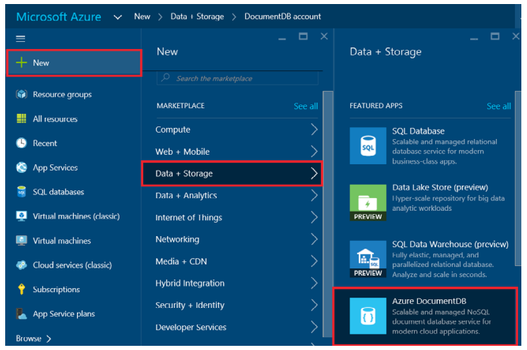
We can create a DocumentDB using the following mechanisms:

- Using Microsoft Azure Portal
- Using **.NET DocumentDB SDK**
- Using REST APIs
- Using JavaScript

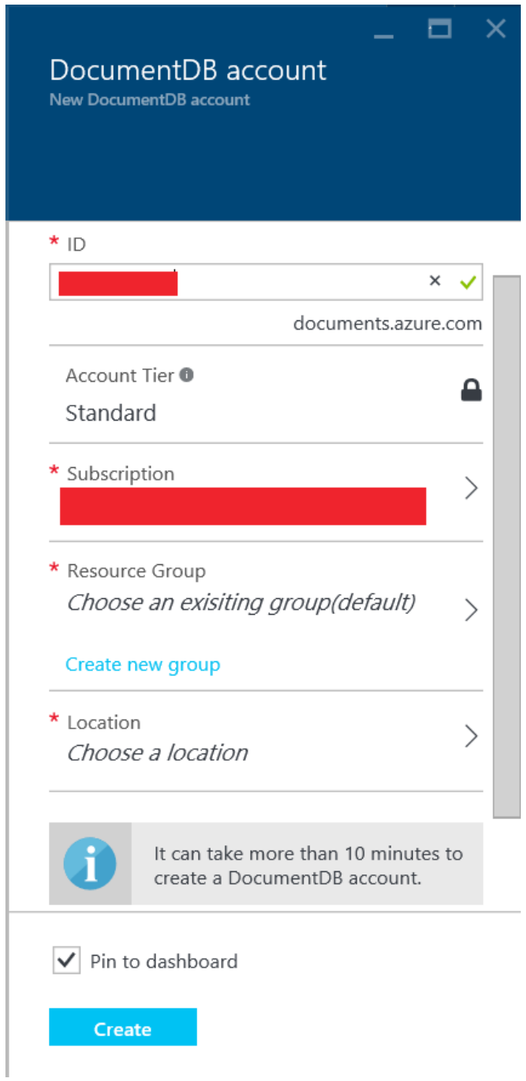
In this article, we will use the .NET DocumentDB SDK. We can use this SDK in a Console Application, WinForm, ASP.NET as well as MVC applications. We can also create a Web API middle-tier and use .NET DocumentDB SDK in it. To explain the concept, the application we will see shortly is implemented using C# Console application and Visual Studio 2015. (Note: This application can be implemented using Visual Studio 2013 as well.)

Let us first create a DocumentDB account using the portal.

Step 1: Visit the Microsoft Azure Portal, log in with your credentials. On this portal, select New > Data+Storage > Azure DocumentDB, as shown in the following image:



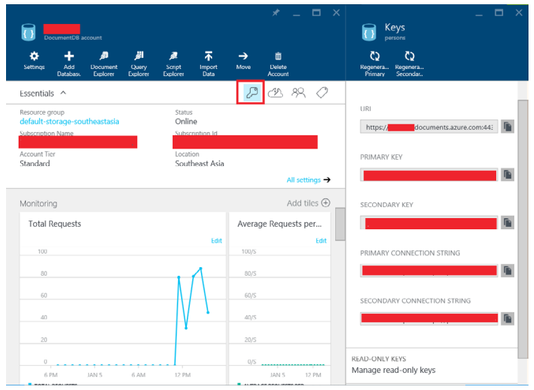
This will open the **DocumentDB Account** panel, enter the account ID in it as shown on the next page:



Now select the **Resource Group**. These are containers which are used to manage a collection of Azure resources. The Location is the data center where the resource will be created. Click on Create button, a new DocumentDB account will be created. We can make use of this account to create Database, Collections etc. in it. The account will be created with the following URL:

```
<account-name>.documents.azure.com
```

Once the account is created, we will need the **URI** information for the account. This is the Endpoint URL using which we can connect to the database for performing operations. We also need a **Master Key.** This is the Primary Key used to authorize the client application to access, create, and query databases. The following image shows the DocumentDB Account Panel from where we can access keys:



Copy these keys, URI and PRIMARY KEY and keep them aside, we will require them in the client application.

Step 2: Open Visual Studio and create a Console application of the name **Demo.DDb.SDKApp**. In this application, add the Microsoft Azure DocumentDB package using NuGet Package Manager. Right click on the **References** and select option **Manager NuGet Packages**. Search for documentdb client and you should see the **Microsoft.Azure.DocumentDB** client library as shown in the following image:

Install this package in the current application. This will add Microsoft.Azure.Documents.Client assembly as well as the Netwonsoft.Json assembly in the project. In the project, add a reference to System.Configuration.dll assembly so that we can read contents from the config file.

What is the DocumentDB Client SDK?

The DocumentDB Client SDK provides classes using which we can directly make use of the DocumentDB client classes to create, access and query DocumentDB Database. This SDK internally manages all HTTP calls.

Step 3: In the App.config file of the console application, add the following appSettings for the DocumentDB Account URL and the Primary Key:

```
<appSettings>
  <add key="DDbEndPoint" value="https://
xxxx.documents.azure.com:443/"/>
  <add key="DDbMasterKey" value="<The
PRIMARY KEY>"/>
</appSettings>
```

Step 4: In the project, add a new class file of the name ModelClass.cs and add the following code in it:

```
using Newtonsoft.Json;

namespace Demo.DDb.SDKApp
{
 public class College
 {
  [JsonProperty(PropertyName = "id")]
  public string collegeId { get; set; }
  public string collegeName { get; set; }
  public string city { get; set; }
  public string state { get; set; }
  public Branch[] branches { get; set; }
 }

 public class Branch
 {
  public string branchId { get; set; }
  public string branchName { get; set; }
  public int capacity { get; set; }
  public Course[] courses { get; set; }
 }

 public class Course
 {
  public string courseId { get; set; }
  public string courseName { get; set; }
  public bool isOptional { get; set; }
 }
}
```

The above entity classes are used to define the College structure. These classes will be used to create document data in the DocumentDB database. The *JsonProperty* attribute applied on the CollegeId property of the College class will define the 'id' for the document.

Step 5: In Program.cs, add the following:

1. The declaration for endpoint url and authorization key. This is used to store information of DocumentDB Account URL and Primary key in it.

```
namespace Demo.DDb.SDKApp

{
class Program
{
 //1.
 static string endpointUrl;
 static string authorizationKey;
```

2. The DocumentClient object - this is the client side logical representation of the DocumentDB Service. The `DocumentCollection` object represents document collection which further contains the `Document` object.

```
static DocumentClient ddbClient = null;
static DocumentCollection docCCollection
= null;
static Document document = null;
```

3. The Main () method contains the logic for reading Endpoint URL and Authorization Key from the App.Config file.

```
static void Main(string[] args)
{

 //3.
 endpointUrl = ConfigurationManager.
 AppSettings["DDbEndPoint"];
 authorizationKey = ConfigurationManager.
 AppSettings["DDbMasterKey"];

 try
 {
  var database =
  createDatabase("CollegesDb").Result;


  createDocumentCollection(database,
  "CollegesInfo");

 }
 catch (Exception ex)
 {
  Console.WriteLine("Error Occured " +
  ex.Message);
 }

 Console.ReadLine();
}
```

4. Call to the `createDatabase()` method.

```
//Method to Create a Database in
DocumentDb
 static async Task<Database>
 createDatabase(string ddbName)
 {
  //4a.
  ddbClient = new DocumentClient(new
  Uri(endpointUrl), authorizationKey);

  //4b.
  Database ddbDatabase = ddbClient.
  CreateDatabaseQuery()
  .Where(d => d.Id == ddbName).
  AsEnumerable().FirstOrDefault();

  if (ddbDatabase == null)
  {
   //4c.
   ddbDatabase = await ddbClient.
   CreateDatabaseAsync(new Database()
   {
    Id = ddbName
   });

   Console.WriteLine("The DocumentDB
   of name " + ddbName + " created
   successfully.");
  }
  else
  {
   Console.WriteLine("This Database is
   already available");
  }
  return ddbDatabase;
 }
```

The code does the following:

- Create the `DocumentClient` object based on the Endpoint url and authorization key. This will work on our Azure subscription where we have created a DocumentDB account.

- This will query over the account to retrieve database under the account.

- If the database, in our case it is CollegesDB, is not available, it will be created.

5. Call to the `createDocumentCollection()` method.

```
static async void
createDocumentCollection(Database ddb,
string colName)

{
 //5a.
 docCCollection = ddbClient.
 CreateDocumentCollectionQuery("dbs/" +
 ddb.Id)
 .Where(c => c.Id == colName).
 AsEnumerable().FirstOrDefault();

 if (docCCollection == null)
 {
  //5b.
  docCCollection = await ddbClient.
  CreateDocumentCollectionAsync("dbs/" +
  ddb.Id,
  new DocumentCollection
  {
   Id = colName
  });

  Console.WriteLine("Created dbs" + ddb.
  Id + "Collection " + colName);

 }
 //6
 createDocumentData(ddb, docCCollection.
 Id, "Colleges");
}
```

This method does the following:

- query the document collection to check if the collection is already present in the database.

- If the collection is not already present, in our case it is **CollegesInfo**, the collection will be created.

6. The `createDocumentCollection()` method calls the `createDocumentData()` method.

```
static async void
createDocumentData(Database ddb, string
collectionName, string docname)
{
 // 6a.
 document = ddbClient.
 CreateDocumentQuery("dbs/" + ddb.Id +
 "/colls/" + collectionName)
 .Where(d => d.Id == docname).
 AsEnumerable().FirstOrDefault();
 if (document == null)
{
 //6b.
 //Record 1
 College msEngg = new College()
 {
  collegeId = "MSEngg",
  collegeName = "MS College of
  Engineeering",
  city = "Pune",
  state = "Maharashtra",
  branches = new Branch[] {
   new Branch() {
    branchId="Mech",branchName=
    "Mechanical",capacity=2,
    courses = new Course[] {
     new Course() {courseId="EngDsgn",
     courseName="Engineering Design",
     isOtional=false },
     new Course() {courseId="MacDw",
     courseName="Machine Drawing",
     isOtional=true }
    }
   },
   new Branch() {
    branchId="CS",branchName=
    "Computer Science",capacity=1,
    courses = new Course[] {
     new Course() {courseId="DS",
     courseName="Data Structure",
     isOtional=false },
     new Course() {courseId="TOC",
     courseName="Theory of Computation"
     ,isOtional=true },
     new Course() {courseId="CPD",
     courseName="Compiler Design",
     isOtional=false }
    }
   }
  }
 };

 //6c.
 await ddbClient.CreateDocumentAsync
 ("dbs/" + ddb.Id + "/colls/" +
 collectionName, msEngg);

 //Record 2
 College lsEngg = new College()
 {
  collegeId = "LSEngg",
  collegeName = "LS College of
  Engineeering",
  city = "Nagpur",
  state = "Maharashtra",
  branches = new Branch[] {
   new Branch() {
    branchId="Cvl",branchName="Civil",
    capacity=2,
    courses = new Course[] {
     new Course() {courseId="EngDsgn",
     courseName="Engineering Design",
     isOtional=false },
     new Course() {courseId="TOM",
     courseName="Theory of
     Mechanics",isOtional=true }
    }
   },
   new Branch() {
    branchId="CS",branchName="Computer
    Science",capacity=3, courses = new
    Course[] {
     new Course()
     {courseId="APS",courseName=
     "Application Software",
     isOtional=false },
     new Course()
     {courseId="TOC",courseName=
     "Theory of
     Computation",isOtional=true },
     new Course()
     {courseId="TNW",courseName=
     "Theory of Computer
     Networks",isOtional=false }
    }
   },
   new Branch() {
   branchId="IT",branchName=
    "Information Technology",
    capacity=3,courses = new Course[] {
     new Course()
     {courseId="DS",courseName="Data
     Structure",isOtional=false },
     new Course()
     {courseId="CSX",courseName=
     "Computer Security
     Extension",isOtional=true },
     new Course()
     {courseId="APS",courseName=
     "Application Software",
     isOtional=false }
    }
   }
  }
 };
 await ddbClient.
 CreateDocumentAsync("dbs/" + ddb.Id +
 "/colls/" + collectionName, lsEngg);

 Console.WriteLine("Created dbs/" +
 ddb.Id + "/colls/" + collectionName +
 "/docs/" + docname);
 }
}
```

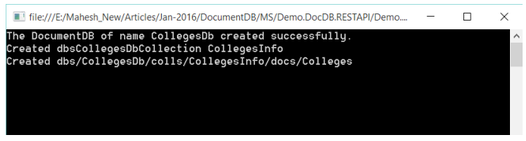
This method does the following:

- The query for the document is executed to check if the document is already present or not. In our case, it is Colleges document.

- If the document is not already present, then College structure is created.

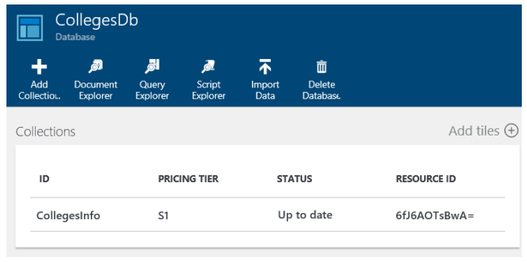
- The College structure is created as document in the collection in the database.

Step 6: Run the application and the following result will be displayed:

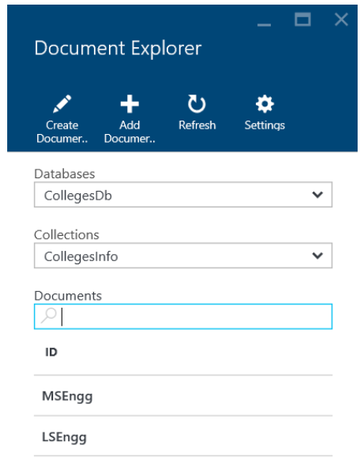


Step 7: To verify the database, collection and document; open the portal and click on the DocumentDB Account name. You will see the CollegesDb database created:

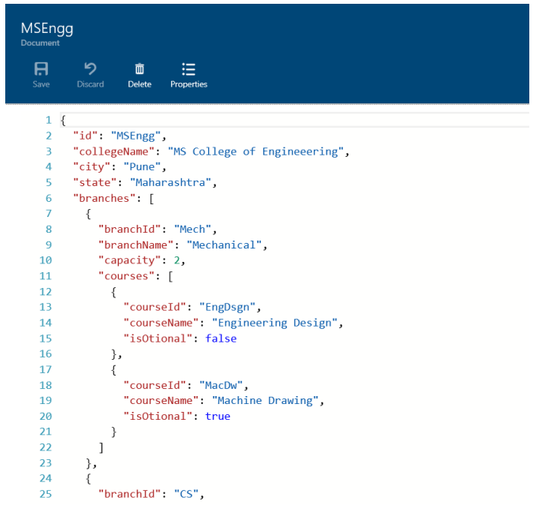
Step 8: Double-click on CollegesDB to view the CollegesInfo collection in it as shown in the following image:



Step 9: Click on the Document Explorer (2nd from left) in CollegesDB. The documents created will be displayed with their names:



Step 10: On clicking on the first GUID, the Document details defined in our code will be shown in JSON format as shown in the following image



The steps so far explained how to create DocumentDB, Collection and document in it. Next, let's see how to query.

Querying DocumentDB

Step 11: In the beginning of the document, we have discussed that DocumentDB can be queried using a SQL like query syntax. In the Program.cs file, add the following method:

```
static void getDatafromDocument(Database
ddb,string collectioName, string docId)
{
 var colleages = ddbClient.
 CreateDocumentQuery("dbs/" + ddb.Id +
 "/colls/" + collectioName,
 "SELECT * " + "FROM CollegesInfo ci " +
 "WHERE ci.id = \"MSEngg\"");

 foreach (var College in colleages)
 {
  Console.WriteLine("\t Result is" +
  College);
 }
}
```

Here we have written a query on the **CollegesInfo** collection for the document MSEngg. The DocumentDB query uses alias similar to SQL statements. Call this method in the Main() method as shown in the following code:

```
//4.
var database =
createDatabase("CollegesDb").Result;

//6.
//
createDocumentCollection(database,
"CollegesInfo");

//8
getDatafromDocument(database,
"CollegesInfo", "MSEngg");
```

After running the application, the following result will be displayed:

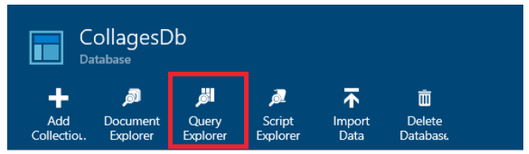
The result shows some additional data properties as follows:

Property	Description
_rid	Resource id
_ts	Time Stamp
_self	URI of the Resource
_etag	GUID used for optimistic concurrency.
_attachments	Attachments if any.

Executing Queries from the Azure Portal

Queries can also be executed directly from the portal.

Step 12: In the portal, click on the Query Explorer of the CollegesDB database as shown in the following image:



Doing so will open the Query Explorer panel with the Database and collection selected. The default select query will be displayed as:

```
SELECT * FROM c
```

Here 'c' means all collections in the database. When the **Run Query** button is clicked, this query will be executed on all collections and result will be returned in a JSON format.

We can write queries as per our requirements. For e.g. Test the following query in the Query Explorer.

```
SELECT * FROM CollegesInfo c where
c.city="Nagpur"
```

This will query College information from a city such as Nagpur.

Conclusion:

DocumentDB provides a schema-free JSON form of data store on Microsoft Azure. This allows us to create database schema, read/write data from and to it. A SQL like syntax for querying data provides an easy feature to manage data.

Download the entire source code from GitHub at bit.ly/dncm23-azure-documentdb

About the Author

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

While writing an application of considerable size in any language, at some point, we need a way to store certain intermediary data in a special property or, a special object. This special property or object shouldn't be easily accessible to the rest of the application to avoid any accidental reassignment of a value.

The most common way to achieve this in JavaScript is by creating a string with current timestamp attached to it. Though we are able to create a unique property name in this way, the data is not secured. The property is easily enumerable. So someone can easily read the value and change it.

The development of Javascript is solely based on ECMA-Script (ES) - a standardized specification of a scripting language. The latest version of ECMAScript is ES6 or ECMAScript 2015.

If you are new to ES6, check out http://www.dotnetcurry.com/javascript/1090/ecma-script6-es6-new-features

METAPROGRAMMING IN ES6 USING SYMBOLS

What is a Symbol and how to create one?

ECMAScript 2015 (ES2015 or, ES6) solves this problem through **Symbols**. Symbol is a new type added to ECMAScript 6 (ECMAScript 2015). It is used to generate a new value on every instance. Value of a Symbol is not shown to the outside world, but the type makes sure that the value is unique. The Symbol object can be used as name of a property or name of a method on an object or a class.

A Symbol can be created using the Symbol factory function. The syntax is shown here:

```
var s = Symbol();
```

It is important to remember that **Symbol() is a function and not a constructor.** An attempt to call it as a constructor would result in an error.

```
var s = new Symbol()  //
error
```

If you try to print value of a Symbol object on the console, you will not see the actual value stored internally, you will see just a Symbol.

```
> var s = Symbol()
<- undefined
> s
<- Symbol()
```

Every call to the Symbol factory function produces a new Symbol. An attempt to compare any pair of symbols results in a Boolean false value.

```
var s1 = Symbol();
var s2 = Symbol();

console.log(s1 === s2);
//false
```

It is possible to create named symbols. We can pass a name as a string parameter to the Symbol factory function.

```
var s = Symbol("mySymbol");
```

But if we use the factory function to create two different Symbols, the two different symbols that will be created, won't be the same.

```
var s1 = Symbol("mySymbol");
var s2 = Symbol("mySymbol");

console.log(s1 === s2);  //false
```

The only way to create a Symbol that can be referred again is using the `Symbol.for()` method. The first call to this method creates and returns a new Symbol with the name provided. Subsequent calls with the same name returns the Symbol that exists in the memory.

```
var s1 = Symbol.for("mySymbol");
var s2 = Symbol.for("mySymbol");

console.log(s1 === s2);  //true
```

When a named Symbol is printed on the console, it displays Symbol(name-of-the-symbol).

Using Symbols

As already mentioned, Symbols can be used to create properties and methods in objects and classes. We can do the following with Symbols:

```
var prop = Symbol();
var method = Symbol();

var obj = {
  [prop]:"Ravi",
  [method]: function(){
    console.log("I am a symbol
    function!!");
  }
};

console.log(obj[prop]); //Prints value
of the property
obj[method]();  //Calls the method
```

Similarly, we can use the above Symbols while creating a class. Following example shows this:

```
var prop = Symbol();
var method = Symbol();

class Sample{
  constructor(){
    this[prop] = "some random value";
  }

  [method](){
    console.log("Method is called....");
  }
}

var obj = new Sample();
console.log(obj[prop]);
console.log(obj[method]());
```

Following are the characteristics of Symbol based members in an object:

- They are not iterable. Meaning, they are not accessible in for...in loop over properties of the object

- They are not serialized when the object is converted to JSON

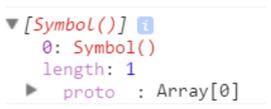
- They aren't included in the list of properties returned by *Object.getOwnProperties* method

Because of this, *Symbols* are almost private properties in objects. However we can still access values of the Symbol properties. We will discuss this in the next section.

Accessing Symbols in an object

There is no direct way to get reference of a Symbol property if we don't already have one. Objects of all Symbol properties can be obtained using the *Object.getOwnPropertySymbols* method. Following snippet gets them and prints the symbols. As you would

have already guessed, the output of the iteration won't make much sense unless the Symbols have names.



As we can see, the result on the console displays only one *Symbol*, it is Symbol of the property. Symbol property of the method is not displayed here.

Using these Symbols, one can iterate over the list and read values of the properties, but meaning of the value won't be known unless the Symbols have meaningful names. ***So the data stored in the Symbol properties is still private in a way, as the values are not directly exposed and are difficult to understand even if someone accesses them***.

Well-known Symbols

ES2015 has a list of predefined Symbols that are used by the JavaScript language for different purposes. They can be used to control the behavior of objects that we create. In this section, we will get to know the available Symbols in ES2015 and how to use them.

Note: *Not all of the following APIs work in all environments that currently support ES2015 to some extent. Check the* ES6 compatibility table *before testing the feature.*

1. Symbol.hasInstance: It is a method that can be used to customize the way *instanceof* operator works with a type. By default, *instanceof* checks if the object is an instance of the type referred and returns false if it doesn't satisfy the condition. We can customize this behavior using `Symbol.hasInstance`. Following is an example:

```
class Person{
 constructor(){
  this.name = "Ravi";
  this.city = "Hyderabad";
 }
 [Symbol.hasInstance](value){
  if(value && typeof(value) !==
  "object" && "name" in value && "city"
  in value)
   return true;

  return false;
 }
}

var p = new Person();

var p2 = {name: "Ram", city: "Chennai"};

console.log(p instanceof Person);
console.log(p2 instanceof Person);
```

Both the console.log statements in the above code block would print *true*.

2. Symbol.toPrimitive: This method can be used to customize the way an object gets converted to a primitive type. JavaScript tries to convert an object to a primitive type when the object is used in an operation with a value of a primitive type.

Some of the common cases where JavaScript tries to convert reference types to primitive types are when operations like addition with a number or, concatenation with a string are performed. If the object is not compatible, the language generates either an error or, produces strange results. The *Symbol.toPrimitive* method can be used to avoid such cases and convert the object to a type based on the value it is operated with.

```
var city = {
  name:"Hyderabad",
  temperature: 40,
  [Symbol.toPrimitive](hint){
    console.log(hint);
    switch(hint){
      case "number": return this.
      temperature;
      case "string": return this.name;
        default: return this;
    }
  }
};

console.log(city*2);
console.log("City is ".concat(city));
```

3. Symbol.toStringTag: This method is used to customize the way `toString` method behaves in an object. A call to `toString` on an object results either in name of the type of which the object is an instance, or just Object if the object is created without using any type. The `toStringTag` method can be used to change the way the method works. Following example shows a usage of it:

```
var city = {
  name: 'Hyderabad',
  currentTemperature: '40'
};


console.log("" + city); // '[object
Object]'

city[Symbol.toStringTag] = function(){
    return this.name;
};

console.log("" + city); // '[object
Hyderabad]'
```

4. Symbol.match, Symbol.replace, Symbol.search and Symbol.split:

The regular expression methods of *String* forward the calls to the corresponding method on their regular expression parameter. The actual implementation of these methods is in the regular *RegExp* class and names of the methods are values of these Symbols.

- String.prototype.match(regExpToMatch) calls regExpToMatch[Symbol.match](this)

- String.prototype.replace(searchPattern, replaceValue) calls searchPattern[Symbol.match](this, replaceValue)

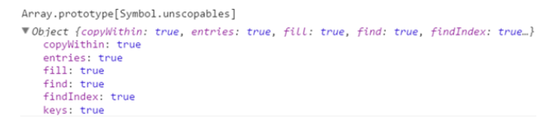
- String.prototype.search(searchPattern) calls searchPattern[Symbol.search](this)

- String.prototype.split(separatorPattern, limit) calls separatorPattern[Symbol.split](this, limit)

5. Symbol.unscopables: Unlike other Symbols, it is an object that is used to hide certain members of an object inside a *with* block.

The *with* operator treats members of an object as variables inside the block. It is not allowed to be used in a strict mode. By default, all members of the object are accessible inside the *with* block.

Symbol.unscopable is used by Array to hide a few of the methods. If you print *Array.prototype[Symbol.unscopable]* in the console, you will see the following:



6. Symbol.species: *Symbol.species* is used to customize the way instances are created in certain circumstances on a type. At times, a class may need to create an instance of itself in a method defined in it. By default, the method creates the instance of the same class. Imagine a scenario where the method is called using an instance of a child class. In such a case, it would be more appropriate to create instance of the child class rather than the current class (which is the parent).

This pattern is used in the *Array.prototype.map* function. It checks for existence of the *Symbol.species* method in the instance used to call the function. If it doesn't exist, it creates instance of the type of the object. Otherwise, it creates instance of the type that is returned by the species function.

Following example shows how `Symbol.species` can be used:

```
class Building{
 constructor(){
  this.floors = 2;
  this.hasGarden = true;
 }

 clone(){
  var hasSpecies = this.
  constructor[Symbol.species];
  var cloneObject = hasSpecies? new
  this.constructor[Symbol.species]() :
  new this.constructor();

  cloneObject.floors = this.floors;
  cloneObject.hasGarden = this.
  hasGarden;

  if(cloneObject instanceof Office){
```

```
    cloneObject.wingsPerFloor = this.
    wingsPerFloor;
   }

   return cloneObject;
  }
}

class House extends Building{
 constructor(){
  super();
  this.rooms = 3;
 }

 static get [Symbol.species]() { return
 Building; }
 }

class Office extends Building{
 constructor(){
  super();
  this.wingsPerFloor;

  this.wingsPerFloor = 2;
 }
}

var h = new House();
h.floors = 3;
console.log(h.clone());

var o = new Office();
console.log(o.clone());
```

In the above example, we have a parent class, `Building`. Two classes inherit from this class, `House` and `Office`. The `clone` method defined in the `Building` class creates a copy of the object and returns it. Out of the two classes, the `House` class has an implementation of the *Symbol.species* static member.

The `clone` method in the `Building` class creates a clone based on presence and return value of this method. The class `House` implements it and returns `Building`. So when the `clone` method is called with an object of `House` type, it returns an object of `Building` type. And when the `clone` method is called with an `Office` object, it returns an object of the same type.

7. Symbol.isConcatSpreadable: This is a property that can be used to configure behavior of the `concat` operation on arrays. When two arrays are concatenated, the second array is spread and its elements are added as individual items in the resultant array. By setting the property `Symbol.isConcatSpreadable` to `false` on the array, we can prevent spreading of the second array and the whole array will be added as an object entry in the resultant array.

```
let arr = [1,2];
let arr2 = arr.concat([3,4], 5);   //
[1,2,3,4,5]


let arr = [1,2];
arr[Symbol.isConcatSpreadable] = false;
let arr2 = arr.concat([3,4], 5);   //
[1,2,[3,4],5]
```

Conclusion

Symbols are added to JavaScript to make the job of creating unique properties easier and also to make a number of default options of the language customizable. They bring in the ability of meta programming to the language. Let's use Symbols and other features of ES6 to make our future JavaScript applications shine.

About the Author

ravi kiran

Microsoft® MVP Most Valuable Professional

Rabi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (ASP.NET/IIS) and DZone MVB awards for his contribution to the community.

SEEDS: INTERFACE SEGREGATION PRINCIPLE, SOLID PART 4

Using interfaces is one of the best ways to allow extensibility and testability in your application. Using interfaces correctly is a key to making this happen well. That's the point of the Interface Segregation Principle (ISP) of SOLID. This principle was created by "Uncle" Bob Martin and states *"Clients should not be forced to depend on methods that they do not use."*

You probably understand what an interface is, but do you know the meaning of segregation? Here in the US when we hear the word segregation, we think of a period of time when children of minority races were sent to their own schools. In other words, they were segregated or set apart.

We should look at a real example of how ISP is used effectively in the .NET Framework. One of the keys to making LINQ work is the IQueryable<T> interface and is defined as:

```
public interface IQueryable<out T> :
IEnumerable<T>, IQueryable, IEnumerable
```

As you can see, IQueryable<T> inherits from three other interfaces. Here's what each interface does according to MSDN:

IQueryable<T>	Provides functionality to evaluate queries against a specific data source wherein the type of the data is known.
IEnumerable<T>	Exposes the enumerator, which supports a simple iteration over a collection of a specified type.
IQueryable	Provides functionality to evaluate queries against a specific data source wherein the type of the data is not specified.
IEnumerable	Exposes an enumerator, which supports a simple iteration over a non-generic collection.

What this shows is that the functionality of iterating over a collection is separated (or segregated) from the functionality of evaluating queries. Furthermore, querying is separated between knowing and not knowing the data type of the data, and iterating is separated between knowing and not knowing the data type of the collection.

When it comes to developing your own classes, think about the specific functionality needed and split that into different interfaces. The book "Adaptive Code Via C#" uses an example of the CRUD (Create, Read, Update, Delete) class. Rather than having a single ICRUD interface for this class, you may want to have ICreate, IRead, IUpdate, and IDelete. This makes it easier to swap out specific functionality, say from reading a database to reading a cache or maybe implementing CRQS, Command Response Query Separation, where commands to read the data are separated from commands to update the data.

This satisfies the principle as classes that inherit the interfaces aren't required to implement functionality they don't need.

You should also beware of the dangers of creating the four interfaces, then having an ICRUD that inherits from each. That doesn't solve the problems you want to avoid as it creates what Uncle Bob calls a "fat interface".

When applying the ISP to software development, we separate interfaces into their functional areas. If this sounds like the Single Responsibility Pattern (SRP), you're onto something.

Where SRP is applied to classes and methods, ISP applies to interfaces.

This principle also enforces high cohesion, giving you better understanding and a more robust class and low coupling, which is more easier to maintain and more resistant to change (ie, less likely to introduce bugs).

Let's turn to an example, first looking at code that violates ISP. This code has been in production for some time to send a message via email.

```
public interface IMessage
{
  IList<string> SendToAddress { get;
  set; }
  string Subject { get; set; }
  string MessageText { get; set; }
  bool Send();
}


public class EmailMessage : IMessage
{
  IList<string> SendToAddress { get;
  set; }
  string Subject { get; set; }
  string MessageText { get; set; }

  bool Send()
  {
    // Contact SMTP server and send
    message
  }
}
```

The team now needs to also send SMS or text messages and decides to leverage the existing interface.

```
public class SMSMessage : IMessage
{
  IList<string> SendToAddress { get;
  set; }
  string MessageText { get; set; }
  string Subject
  {
    get { throw new
    NotImplementedException(); }
    set { throw new
    NotImplementedException(); }
  }

  bool Send() {
    // Contact SMS server and send
    message
  }
}
```

Because SMS doesn't have a Subject, an exception is thrown. You can't simply take out Subject because it's required by the interface. It can get worse if the team decides to add CCToAddress.

```
public interface IMessage
{
 IList<string> SendToAddress { get;
 set; }
 IList<string> CCToAddress { get; set; }
 string Subject { get; set; }
 string MessageText { get; set; }
 bool Send();
}

public class SMSMessage : IMessage
{
 IList<string> SendToAddress
 { get; set; }
 string MessageText { get; set; }
 string Subject
 {
  get { throw new
  NotImplementedException(); }
  set { throw new
  NotImplementedException(); }
 }

 string CCToAddress
 {
  get { throw new
  NotImplementedException(); }
  set { throw new
  NotImplementedException(); }
 }

 bool Send()
 {
  // Contact SMS server and send message
 }
}
```

It would get even worse with BCCToAddress and email attachments. A better way is to put the interface on a diet and have it comply with the Interface Segregation Principle.

```
public interface IMessage
{
 IList<string> SendTo { get; set; }
 string MessageText { get; set; }
 bool Send();
}

public interface IEmailMessage
{
 IList<string> CCTo { get; set; }
 IList<string> BCCTo { get; set; }
 IList<string> AttachementFilePaths
 { get; set; }
 string Subject { get; set; }
}

public class EmailMessage : IMessage,
IEmailMessage
{
 IList<string> SendTo { get; set; }
 IList<string> CCTo { get; set; }
 IList<string> BCCTo { get; set; }
 IList<string> AttachementFilePaths {
 get; set; }
 string Subject { get; set; }
 string MessageText { get; set; }

 bool Send()
 {
  // Contact SMTP server and send
  message
 }
}

public class SMSMessage : IMessage
{
 IList<string> SendTo { get; set; }
 string MessageText { get; set; }

 bool Send() {
  // Contact SMS server and send message
 }
}
```

So, put your interfaces on a diet. Don't make them fat. By doing so, you will allow your software to grow and thrive and be lush, green and vibrant. In the next issue, I'll finish up the discussion of SOLID.

About Software Gardening

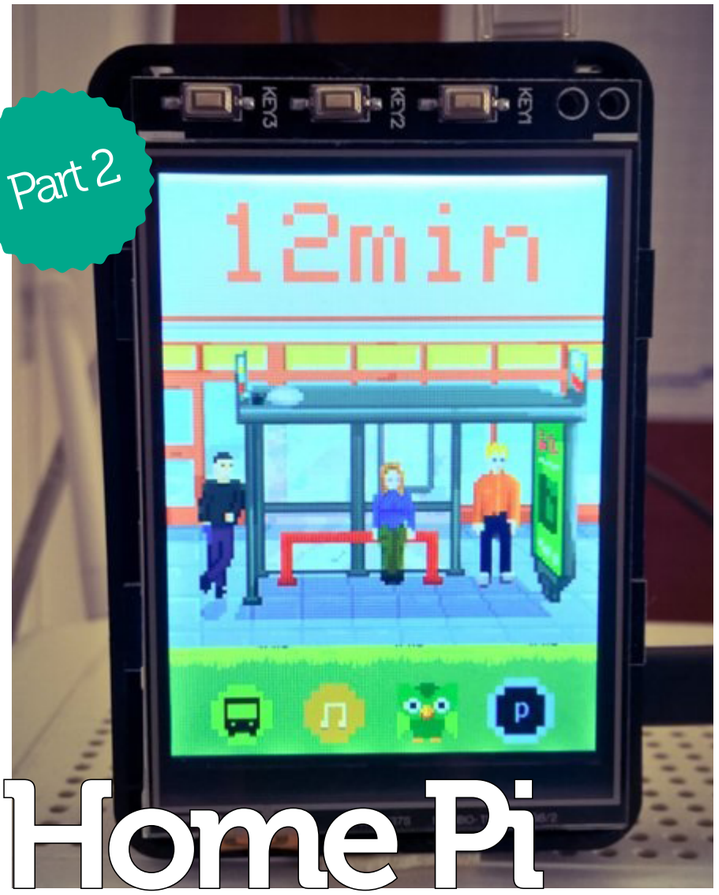
Comparing software development to constructing a building says that software is solid and difficult to change. Instead, we should compare software development to gardening as a garden changes all the time. Software Gardening embraces practices and tools that help you create the best possible garden for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

About the Author

craig berntson

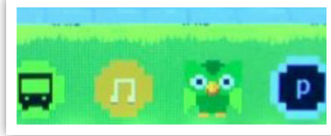
MVP Microsoft® Most Valuable Professional

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.



This article is an extension to the first part HomePi - A Windows 10 & Raspberry Pi 2 IoT app [bitly.com/win10pi2]

Previously, we looked at the h/w required, project structure, Display/Touch drivers and also added a small code snippet to display current weather information. This is part two of our quest to build our Home Pi Windows 10 IoT project. In this series, we will add the remaining features and make our app interactive. Shown above is the complete app displaying the next Bus schedule.



Let us jump into the code right away. To get some background of the app we have built so far, visit http://www.dotnetcurry.com/windowsapp/1240/homepi-windows-10-iot-app-raspberrypi-2

Project

MainPage.xaml

Add the following code to the main Grid:

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

In the above code, we have added a new Media Player element and this will be used for Audio play back. **MediaEnded** event will automatically play the next track once the current track is complete.

Even though Home Pi has four screens (Next Bus, Weather, Music Player and Power Control), we will use a single XAML file to manage all four screens. Add the following enum values to the class.

```
enum CurrentPage
{
  Weather,
  NextBus,
  MusicPlayer,
  Power
};
```

Whenever a menu icon is pressed, we will use the enum values to identify the current page and display information accordingly.

This page also has methods that are detailed here. Please check bitly.com/win10pi2 along with the source code to understand these methods in detail.

Timers

appTimer: This timer is used to display the count down for the next bus, and runs every minute.

touchTimer: This timer executes every 50 milli seconds and constantly checks for Touch Inputs.

TouchTimer_Tick

```
private void TouchTimer_Tick(object
sender, object e)
{
  TSC2046.CheckTouch();

  int x = TSC2046.getTouchX();
  int y = TSC2046.getTouchY();
  int p = TSC2046.getPressure();

  if (p > 5)
  {
    CheckAction(TSC2046.getDispX(),
    TSC2046.getDispY());
  }
}
```

This method gets the x, y coordinates and the pressure value of the touch input using `getTouchX()`, `getTouchY()` and `getPressure()` methods in Touch processor class (refer previous article about Touch and Display processors). To avoid processing accidental touch inputs, only pressure values greater than 5 are processed and rest are ignored. `CheckAction()` method is used to process the touch input. This process is repeated every 50 milli seconds.

CheckAction

```
private void CheckAction(int x, int y)
{
 Windows.Foundation.Point touchPoint =
 new Windows.Foundation.Point(x, y);
 Rect rect;
 bool isControlpoint;

 switch (currenPage)
 {
  case CurrentPage.Power:
  rect = new Rect(64, 53, 108, 110); //
  Power button is within this rectangle
  area
  isControlpoint = rect.
  Contains(touchPoint);
  if (isControlpoint)
  {
   ShutDown();
  }
   break;
 }

 //Menu controls
 rect = new Rect(275, 20, 40, 40); //
 Next bus Icon is within this rectangle
 area
 isControlpoint = rect.
 Contains(touchPoint);
 if (isControlpoint)
 {
  GetNextBus();
 }

 rect = new Rect(275, 72, 40, 40);   //
 Music player Icon is within this
 rectangle area
 isControlpoint = rect.
 Contains(touchPoint);
 if (isControlpoint)
 {
  PlayMusic();
 }
 rect = new Rect(275, 125, 40, 40); //
 Weather Icon is within this rectangle
 area
 isControlpoint = rect.
 Contains(touchPoint);
 if (isControlpoint)
 {
  GetWeather();
 }

 rect = new Rect(275, 179, 40, 40);
 isControlpoint = rect.
 Contains(touchPoint); //Power button is
 within this rectangle area
 if (isControlpoint)
 {
  ShowPowerPage();
 }

}
```

If the input pressure is greater than 5, then this method is called. Next step is to check if touch

point falls under any of the menu controls. Menu items are nothing but just one whole Image in the project. Basic idea is to check if the x and y points lie somewhere within each x and y points of every pixel in the image. One way is to iterate through all pixels within the menu area and then compare it with x and y. A simpler approach is to use the inbuilt .NET classes and this can be achieved using the `Rect` structure and `Contains` method. I used Photoshop to find out the x, y coordinates of each menu icon and used `Rect` structure to build a Rectangle area. Then used it to check if the touch point lies within this area using `Contains` method. Inputs in any other areas are ignored:

Shutdown

```
private void ShutDown()
{
    ShutdownManager.
BeginShutdown(ShutdownKind.Shutdown,
TimeSpan.FromSeconds(0.5));
}
```

This method shuts down Raspberry Pi using the `ShutdownManager` class.

PlayMusic

```
private async void PlayMusic()
{
 currenPage = CurrentPage.MusicPlayer;

 //paint Menu
 await ILI9341.LoadBitmap(display1, 0,
 0, 240, 320,
  "ms-appx:///assets/Music_Home.png");
 likes = await Utilities.GetLikes();
 if (likes.Count > 0)
 {
  LoadTrack(likes[nowPlaying]);
 }
}
```

`PlayMusic()` method loads a list of likes from Sound Cloud and starts playing the first track in the list. `Utilities.GetLikes()` method is explained shortly.

GetNextBus

```
private async void GetNextBus()
{
 currenPage = CurrentPage.NextBus;
 await ILI9341.LoadBitmap(display1, 0,
 0, 240, 320, "ms-appx:///assets/Bus_
 Home.png");
 nextBus = await Utilities.GetNextBus();
 ILI9341.setCursor(display1, 30, 15);
 string s = nextBus.ToString() + " min";
 ILI9341.write(display1,
 s.ToCharArray(), 6, 0xDB69);
}
```

This method sets the `currenPage` value, loads a new background Image and gets the schedule for next bus in minutes to displays it on screen. Number of minutes is updated every minute using AppTimer which runs every minute.

AppTimer_Tick

```
private async void AppTimer_Tick(object
sender, object e)
{
 //Run only if the current page is Next
 Bus
 switch (currenPage)
 {
  case CurrentPage.NextBus:
  //Erase screen
  ILI9341.fillRect(display1, 0, 0, 240,
  70, 0xC616);
  nextBus -= 1;
  if (nextBus <= 2)
  {
   nextBus = await Utilities.
   GetNextBus();
  }

  ILI9341.setCursor(display1, 30, 15);
  string s = nextBus.ToString() + "
  min";
  ILI9341.write(display1,
  s.ToCharArray(), 6, 0xDB69);
  break;
 }
}
```

This method executes every minute and updates the minute value. This is done only if the current page is `nextBus`. To avoid reaching API limits with my local transport provider, I check for updates only if the next bus arrives in less than 2 minutes. Why 2 minutes and not the last minute? It takes two minutes to walk to my nearest bus stop ;)

ShowPowerPage

This method displays an image with a Power icon and touching the power icon will shut down the Raspberry Pi using `ShutDown()` method. CheckAction() method takes care of processing touch input and checks if it lies within the Power Icon area.

LoadTrack

```
private void LoadTrack(SoundCloudTrack
currentTrack)
{
 //Stop player, set new stream uri and
 play track
 mPlayer.Stop();
 Uri streamUri = new Uri(currentTrack.
 stream_url + "?client_id=YOUR_CLIENT_
 ID_HERE");
 mPlayer.Source = streamUri;
 mPlayer.Play();
}
```

This method loads a new track in the Media Player and starts playing the current track.

mplayer_MediaEnded

```
private void mPlayer_MediaEnded(System.
Object sender, RoutedEventArgs e)
{
 nowPlaying += 1;
 if (nowPlaying > likes.Count) //Reset
 to first track
 nowPlaying = 0;
 LoadTrack(likes[nowPlaying]);
}
```

This method gets called automatically when the track has finished playing. Global counter `nowPlaying` is updated and next track is loaded. It will also reset the counter to 0 to start playing again from the first track, if there are no more tracks to play.

Utilities.cs

This file has some extra methods and classes that are used for Sound Cloud and Next bus service calls.

GetNextBus

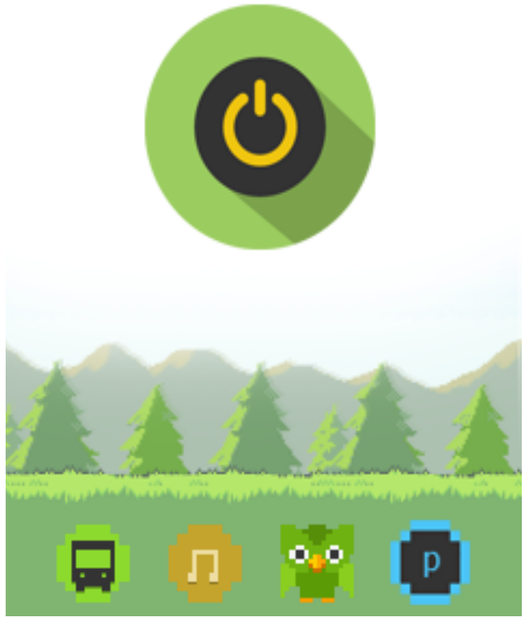
```
public static async Task<int> GetNextBus()
{

 int nextbusIn = 0;
 try
 {
  var response = await GetjsonStream
  ("http://www.wienerlinien.at/
  ogd_routing/XML_TRIP_REQUEST2?
  locationServerActive=1&outputFormat=
  JSON&type_origin=stopid&name_origin=
  60200884&type_destination=
  stopid&name_destination=60200641");
  RootObject obj =
  JsonConvert.DeserializeObject
  <RootObject>(response);

  foreach (Trip trip in obj.trips)
  {
   BUSDateTime nxt = trip.trip.legs[0].
   points[0].dateTime;
   DateTime nextBus = DateTime.
   ParseExact(nxt.time,
   "HH:mm", System.Globalization.
   CultureInfo.InvariantCulture);

   if (DateTime.Compare(DateTime.Now,
   nextBus) == -1) //Return the first
   next bus
   {
    TimeSpan diff = nextBus -
    DateTime.Now;
    int minutes = Convert.ToInt16
    (Math.Round(diff.TotalMinutes, 0));
    if (minutes > 2)
    {
     nextbusIn = minutes;
     return nextbusIn;
    }
   }
  }
 }
 catch (Exception)
 {
  //Handle error
 }
 return nextbusIn;
}
```

In this method, I make a call to my local transport API (Vienna, Austria) to get a list of buses from one point to another. I have hard coded the values for origin and destination. This API returns the next 5 buses as JSON string and I use Newtonsoft JSON

library to get the first next bus and check if it arrives in less than 2 minutes. If so, return the second bus. Remember, it takes 2 minutes to walk to my nearest bus station. You can replace this with your own transport API or any other service for which you need a countdown for.

GetLikes

This method returns a list of favourites from your Sound Cloud profile. This is a simplified version of Music Player detailed in Build a Windows 10 Universal App – First Look



Future Enhancements

Home Pi App is good enough for a hobby project but you can make it better with a few suggestions:

1. Better error handling
2. Restart option in Power Page
3. More media player controls for Music Player
4. Add different UI to MainPage.xaml so that you can have different UI for HDMI output and LCD screen.

Download the entire source code from GitHub at bit.ly/dncm23-win10-pi2-iot

About the Author

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*

AZURE ACTIVE DIRECTORY ESSENTIALS FOR ASP.NET MVC DEVELOPERS

Azure Active Directory has emerged as a complete package for satisfying your application's "Identity Management" needs.

Be it the requirement of implementing Single SignOn(SSO) using on premises identity, Cloud only identity, Federation (or authentication) against Cloud SaaS applications (like Office 365, Salesforce, Dropbox, Facebook at work etc.), or the new age identity requirements like Multi factor authentication, Self-service management scenarios, Bring Your Own Device (BYOD) scenarios and device registrations; everything is possible today with Azure Active Directory. This step by step article will take you through the basics of Azure Active Directory and demonstrate the most common task of authenticating an ASP.NET MVC application using Azure Active Directory.

Introduction

In this article, we will explore the following important aspects of Azure Active Directory –

1. Terminologies and Concepts

2. Comparison with Windows Server Active Directory at conceptual level

3. Step by step guide to secure an ASP.NET MVC application using Azure AD with -

a. Windows Identity Foundation
b. OWIN KATANA (this is the future).

Let's get started!

Applicable Technology Stack

Here are the pre-requisites for this article:

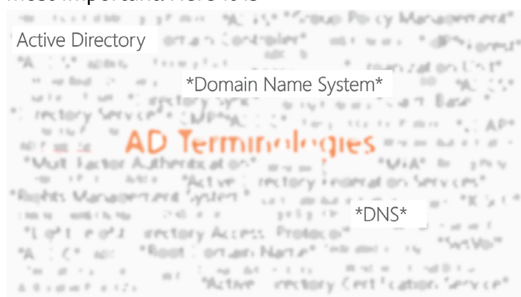
1. Valid Microsoft Azure subscription. Free trial is available here.

2. Visual Studio 2013 with update 5 Or Visual Studio 2015 or the Free Visual studio Community Edition.

Terminologies and Concepts

Since its inception, Azure Active Directory has become the backbone to most of the services offered by the Microsoft Azure platform. Most of the times, whenever a developer hears about Active Directory, he/she gets nervous as a result of the huge number of features and terminologies involved with this concept. Ironically, once I tried to list down only the very important terminologies used in context of Active Directory in one PPT slide and this what I ended up with –

When you look at the above picture, how you do feel. Excited or Overwhelmed? The first thought that comes to our mind is why so much trouble? Let us filter through this plethora of huge bulky terminologies and focus on the ones we feel are the most important. Here it is –



This is our mental picture of Active directory and its terminologies. We always refer to these 3 words when we talk about Active directory. So let's first brush up on these basic, but important Active Directory terminologies.

Directory Service

> In layman's term, a directory is like a book containing alphabetical index of the names and addresses of people in a city or an area. Telephone directory was a popular form of directory. If we talk in technical terms, it is a **mapping between names and values.** If you see a Word Dictionary like Oxford, for each word, you will find many different meanings and definitions.

In case of software systems, the basic communication language is English (pun intended). Well it is not English, it is TCP/IP. So when any computer participates in TCP/IP communication, a numerical label is assigned to each computer and we call it **IP address** of the machine. Soon due to explosion of internet connectivity and desktop PCs; people found it difficult to remember the IP addresses of the computer to communicate with, therefore they decided to provide some **user friendly names**, something a user can easily remember. But again there were so many user friendly names required to be given, that mapping of those user friendly names to IP address number was a challenge to remember. Therefore a system was developed which remembers or maps the user friendly name to IP address so that users are freed from remembering this association.

This is called as **Directory Service**.

Domain Name System

So "Directory Service" is the same as telephone directory, but for computer systems, it has names and associated multiple IP address values in it. Every Directory Service contains many computers so to denote that these group of computers are part of this directory service and each computer within directory service, will have a unique numerical value which we call as IP address.

Directory service is a *concept* and many implementations were carried out from this concept. The most popular implementation is **Domain Name System**, commonly known as **DNS**. This implementation is very popular and is still used today for Internet and Private Network (organizations). Therefore directory service is commonly known and referred as DNS.

Windows Domain, AD and ADDC

What you see on the next diagram is the hierarchy of Directory Service and where DNS fits in. Similar to DNS, some other systems have been developed like NIS, NT domain and many more but they are not popular and are rarely used.

Directory service standards was well suited for internet services but there were some shortcomings, issues and security problems while implementing DNS for a private organization network. Therefore those standards were reformed and named as **X.500.**

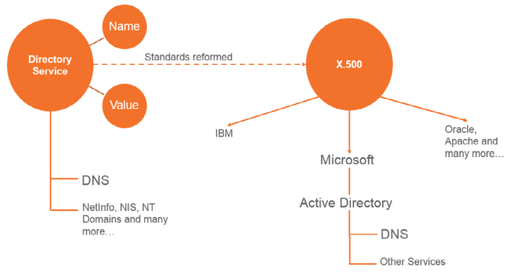
> *I don't know why but people always love giving absurd names to standards. X.500 represents a fantastic example of the same. No offence please, I am just expressing my disbelief! Or possibly I am dumb enough not able to understand the reasons behind this and many more naming conventions.*

These standards give you a way to develop a directory for private organization. So there are multiple implementations of X.500.

Now Microsoft created a special type of computer network based on TCP/IP in which, user accounts, computer info, printers, security principles can be stored on computers itself. This special computer network was called as **Windows Domain**. For this domain, Microsoft felt the need of Directory service to do the mapping. Therefore Microsoft also did the implementation of X.500 standards and came up with a product. ***This product was called as Active Directory.***

> *So AD is a directory service for Windows Domain. The computers used for storing the information of Windows domain in Active directory are called as* **Domain Controllers or ADDC**.

These AD based systems also had a requirement to connect to the internet, therefore DNS system was made a part of AD. So essentially AD and DNS are two different things and to support internet based connectivity for AD based resources, DNS was made a part of it.

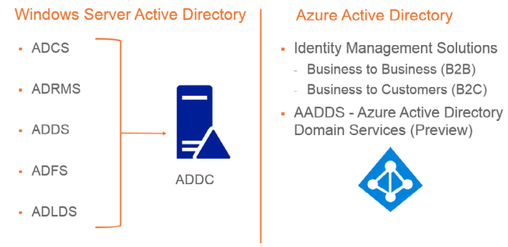


So this is the general background of Active directory and how it came into existence. Does it make sense to you? Cool!

Now that we understood what active directory is, let's understand what Azure Active Directory is by comparing them.

Comparing Windows Server AD and Azure AD

When you hear about Azure Active directory and provided you have background of Windows AD; the first thought that comes to mind is "Windows AD and Azure AD must be same with Azure AD being cloud offering". Unfortunately, this is far from being true. Let's see why.



Windows Active Directory offers 5 core services and they are –

ADCS – Active Directory Certification Services – Offers public key certificates if you wish to implement certificate based authentication.

ADRMS – Active Directory Rights Management System – It allows to create policies that enforce control over access to information. The idea of RMs is unique and protection provided to any piece of information is Persistent. This means protection rules travels with data and can't be separated.

ADDS – Active Directory Domain Services – Allows to manage network resources such as computers and users within network and integrates with DNS service of Active Directory.

ADFS – Active Directory Federation Services – Used for federating the identity of user and based on WSFederation standards.

ADLDS – Active Directory Light Directory Services – Similar to ADDS. Just based on LDAP standards.

All of these services are installed or configured on one server to which we usually refer to as Active Directory Domain Controller (ADDC).

On the other hand, **Azure AD offers you 2 key solutions only** –

1. Identity management solution.

2. Azure AD Domain Services (AADDS - in preview as of today).

Windows Server AD is not an identity solution but Azure AD is. Window server Active directory is designed to operate in corporate networks where enterprise has full control over topology. This topology exposes services to manage –

- Identity
- Servers
- Workstations
- Security
- Network policies and so on.

And these are exposed through enterprise protocols like LDAP, Kerberos, DNS, AD replication and so on.

On the other hand, Azure AD is focused only on one area – Identity throughout the internet. Here –

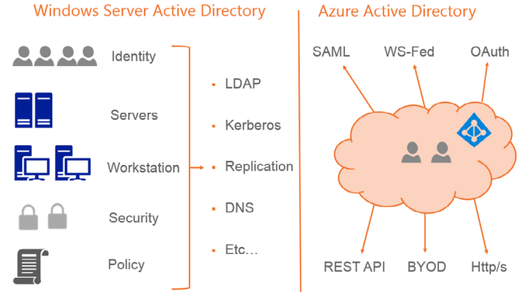
- Types of communication is http and https.

- Types of devices are different and not only corporate assets

- Protocols are SAML, WS Federation, OAuth, OpenId

- And instead of LDAP, you talk here about REST API.

This ability of Azure AD enables us to make application hosted as Software as a Service (SaaS) or even connect to existing many enterprise grade SaaS applications like SalesForce, Office 365, Box, Facebook at work and so on.



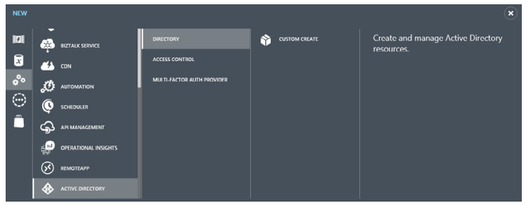
How to create Azure Active Directory?

As of today, most of the services can be used from the new Azure portal (http://portal.azure.com).

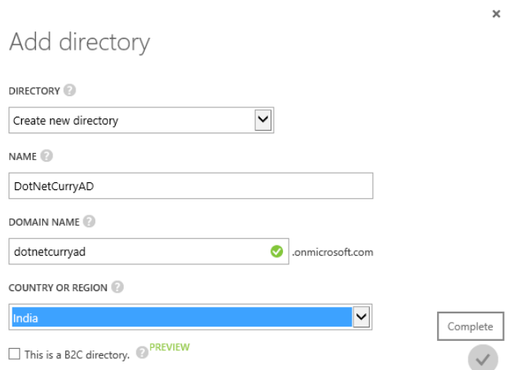
However there still are some services for which we have to fall back to the full management portal (http://manage.windowsazure.com/). As of this writing, Azure AD can be managed using the old portal only.

If we click on the Active directory option, by default you will see at least one directory already present. This is the default active directory under which your Azure subscription is present and one directory can have multiple subscriptions within it. To keep things simple, let's first create a new directory itself.

Click on New > App Services > Active Directory > Directory > Custom Create as shown below –



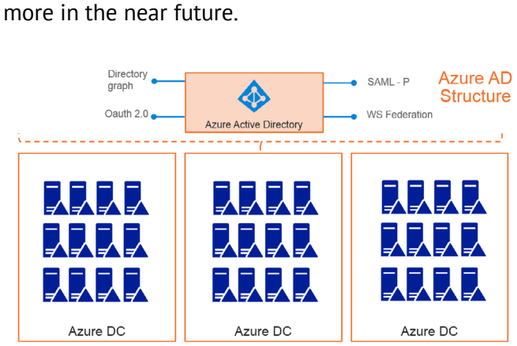
Provide the name of the directory as "DotNetCurryAD" and rest of the values as shown in the image below. Of course you may choose the country of your choice. Then click on *Complete* to create an Azure AD.



Open the created Azure AD and click on "Users" tab. The current logged in user automatically becomes "global administrator" in the newly created directory, giving the user full access to the entire Azure active directory. Various admin roles in Azure AD and their descriptions are well documented at http://bit.ly/1pb1V5C . With this, let's go back to understand some more concepts.

Structure of Azure Active Directory

If you have observed the Fully Qualified Domain Name (FQDN) of the AD we created, it is **dotnetcurryad.onmicrosoft.com**. From this name, **onmicrosoft.com** is the shared domain. This means Microsoft.com already exists and we are using a part of it. So as you can see in the following diagram, **Azure active directory is a directory service that lives on the cloud**. It is highly scalable, distributed service that runs across many servers in data centers of Microsoft, across the world. And most importantly, it is offered as a Platform as a Service (PaaS) offering. What does that mean? You don't have direct access to underlying servers, but the hardware and software is delivered to you as a service. But then the next question that comes to mind is how do we access it? As "endpoint and interfaces". So the service exposes interfaces such as directory graph API; and authentication endpoints such as OAuth, SAML, WSFederation and a couple of more in the near future.



Important: This active directory service contains the data which is owned by each customer who is a tenant of the directory. Pay attention to "Tenant of the directory". Let's understand more about it in the next section.

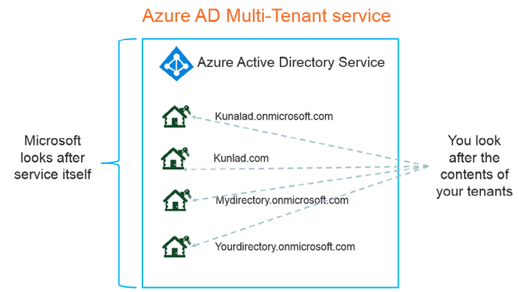
Why Azure AD is called as Multi-Tenant Service?

The dictionary meaning of a tenant is a person who hires a property on rent. Similarly you may own an office space in a building. This building may be shared by multiple offices. So your office can be considered as a tenant of the building. This building provides all the services to carry out your business well, including security. It also ensures you are isolated from other organizations hosted in the same building.

In Azure, Active Directory service is exactly similar to this concept. It contains many such tenants. So if we want to give it a fancy name, we can say Azure active directory is **Multi-Tenant service**.

So coming back to the statement we made earlier, "this active directory service contains the data which is owned by each customer who is a tenant of the directory".

There are lots of tenants inside the service. These tenants have names like KunalAD.onmicrosoft.com. But they can also be configured for kunalad.com. Each customer looks after their data in each tenant, whereas Microsoft looks at the entire Azure Active directory service to keep it up and running all the time. And then you manage, change, update the data using portal. So when you hear the term saying Azure AD is multi-tenant service, this is what it exactly means.



Alright, I have tried my level best to explain Azure AD concepts using the simplest language and diagrams; and now is the time for some action. Let's build an application which can authenticate against Azure AD.

Secure ASP.NET MVC application using Azure Active Directory

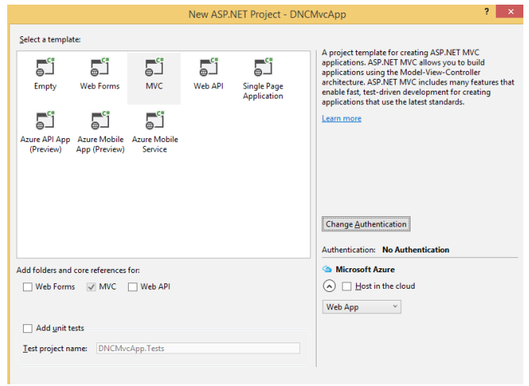
There are 2 popular implementation approaches used for securing ASP.NET MVC application using Azure active directory –

1. Using OWIN KATANA
2. Using Windows Identity Foundation (WIF)

We will build the application using both the techniques and with the "Ws-Federation" protocol which is quite common. Let's start with OWIN first.

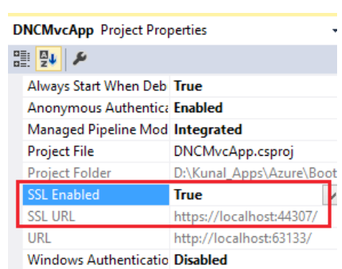
Securing MVC application using Azure Active Directory and OWIN KATANA

We will build our entire application from scratch. Create a simple ASP.NET MVC application using Visual Studio. Name the application as "DNCMvcApp". Select the template as MVC, authentication as "No Authentication" and uncheck the checkmark of "Host in the cloud" for now. Click Ok to proceed ahead and create the application.

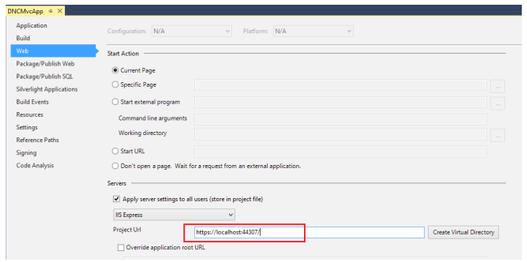


Configuring Https

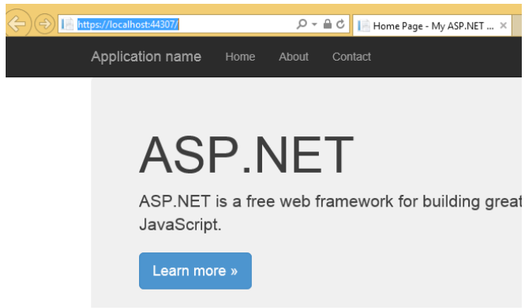
We need to configure the project to run on https. Therefore open the properties by selecting the project from Solution Explorer and by pressing "F4", and mark the property "SSL Enabled" as true and copy the new URL configured under property "SSL URL".



Now open the project properties by right clicking on the project from Solution Explorer. Paste the copied URL to web properties as shown here –



Build and run the application. You may be prompted by a certificate error which you can safely ignore and proceed ahead. This error is because of the self-signed certificate. Make sure that application runs on the same URL which in my case is https://localhost:44307/ as shown here.



Add important nuget packages

Open "Package manager console" and run the following commands to add required references to the application.

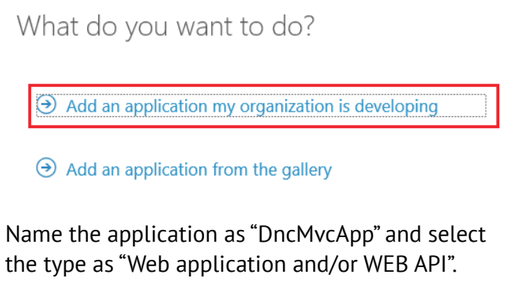
Install-Package Microsoft.IdentityModel.Protocol.Extensions -Version 1.0.2.206221351	Configures hosting prerequisites to ensure that OWIN initialization gets picked up at launch time.
Install-Package System.IdentityModel.Tokens.Jwt -Version 4.0.2.206221351	
Install-Package Microsoft.Owin.Security.WsFederation	Ws-Federation package to enable same for authentication purpose
Install-Package Microsoft.Owin.Security.Cookies	Cookie middleware for managing the sessions
Install-Package Microsoft.Owin.Host.SystemWeb	

Rebuild the application to check if it builds successfully.

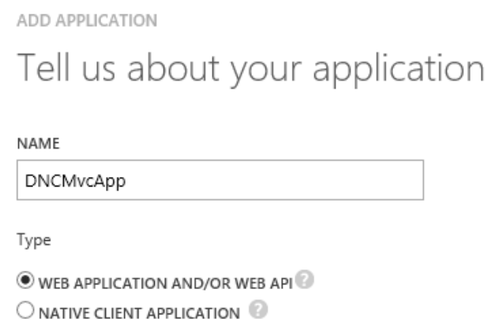
Register application in Azure Active Directory Tenant

To secure our DNCMvcApp using Azure AD, our application should trust Azure AD. Similarly Azure AD will need to know about our application to trust it. This can be achieved by simply adding the application in our Azure Active Directory Tenant "dotnetcurryad.onmicrosoft.com".

Open Azure AD from the Azure portal and click on the Applications tab. Select the option "Add an application my organization is developing".



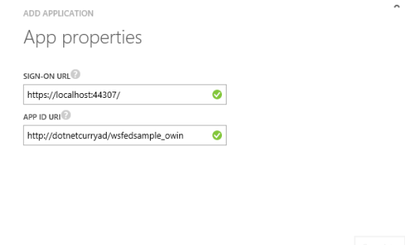
Name the application as "DncMvcApp" and select the type as "Web application and/or WEB API".



The *type* would of course be *web*. If I want my desktop application authenticated against Azure AD, then we need to select the Native Client Application option. On the next screen, we need to provide SignOn URL and App Id URI. The application URL is https://localhost:44307/ and this is nothing but our signon url. This is important, **your application url must match the sign on url.** Therefore we will put the same https url of our application copied from project properties, in here.

The App Id URI is the identifier for the application, or if I have to put this in fancy words, **in WS-Federation parlance, it is the realm of the application**. You can choose any identifier that makes sense to you, as long as it is unique within the active directory tenant. I usually try to follow a memorable schema. **Remember, given that this is a URI, this value does not need to correspond to any real network address!** And of course here I assume you understand difference between URI and URL. If not refer to identity guru Vittorio's article (http://bit.ly/1TId5vc) for differences between URL, URI and URN.

We put the value as http://dotnetcurryad/wsfedsample_owin. This is an Owin WsFed sample and under DotNetCurryAD tenant. Click OK to proceed.



This completes the configuration on Azure AD side. Let's go back to Visual Studio.

Add and configure Startup class

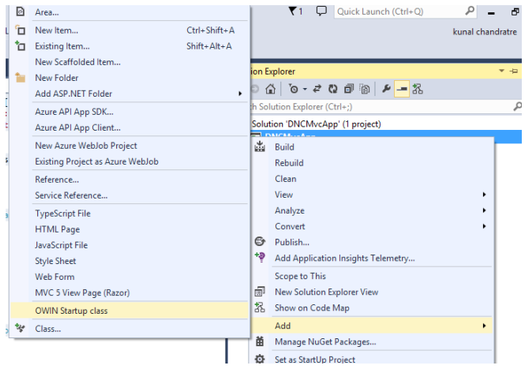
Startup.Auth.cs is a very important class and is used for OWIN initialization. In Visual Studio, right click on App_Start > Add > New Item > Class. Name the class file as Startup.Auth.cs. Copy paste the following code segment in the same class –

```
namespace DNCMvcApp
{
public partial class Startup
{
//
// Client ID is used by the
application to uniquely identify
itself to Azure AD.
// Metadata Address is used by the
application to
// retrieve the signing keys used by
Azure AD.
// The AAD Instance is the instance of
Azure, for example public Azure or
Azure China.
// The Authority is the sign-in URL of
the tenant.
// The Post Logout Redirect Uri is the
URL where the user will be redirected
after they sign out.
//

private static string realm =
ConfigurationManager.AppSettings
["ida:Wtrealm"];
private static string aadInstance =
ConfigurationManager.AppSettings
["ida:AADInstance"];
private static string tenant =
ConfigurationManager.AppSettings
["ida:Tenant"];
private static string metadata =
string.Format("{0}/{1}/
federationmetadata/2007-06/
federationmetadata.xml", aadInstance,
tenant);

string authority = String.Format
(CultureInfo.InvariantCulture,
aadInstance, tenant);

public void ConfigureAuth
(IAppBuilder app)
{
app.SetDefaultSignInAsAuthenticationType
(CookieAuthenticationDefaults.
AuthenticationType);

app.UseCookieAuthentication
(new CookieAuthenticationOptions());

app.UseWsFederationAuthentication(
new WsFederationAuthenticationOptions
{
Wtrealm = realm,
MetadataAddress = metadata,
Notifications = new
WsFederationAuthenticationNotifications
{
AuthenticationFailed = context =>
{
context.HandleResponse();
context.Response.Redirect
("Home/Error?message=" +
context.Exception.Message);
return Task.FromResult(0);
}
}
});
}
```

```
}
}
```

Remember Owin is a specification and KATANA is Microsoft's implementation of this specification. The code we just saw is a standard code a startup class can have. OWIN basics and startup class is already covered in a previous article of authenticating ASP.NET MVC using Owin Katana and ADFS [link - http://www.dotnetcurry.com/windows-azure/1166/aspnet-mvc-multiple-adfs-owin-katana] so let's not spend more time on this. The code is quite self-explanatory with the comments.

Now we will add an OWIN startup class and invoke the ConfigureAuth() method of startup.auth.cs class. Right click on project > Add > Add OWIN Class. Provide the name as **Startup.cs**. Note, earlier the class file was Startup.Auth.cs, whereas this class is named as Startup.cs.



Replace the code with the following –

```
public partial class Startup
{
 public void Configuration(IAppBuilder
 app)
 {
  ConfigureAuth(app);
 }
}
```

So essentially we are just invoking ConfigureAuth() method in this class. Here we complete the startup class configuration.

Configure Login page

In Views > Shared folder, right click on Add New Empty View. Name it as _LoginPartial.cshtml. Replace the code in this file with the one given here –

```
@if (Request.IsAuthenticated)
{
 <text>
 <ul class="nav navbar-nav navbar-
 right">

 <li class="navbar-text">
 Hello, @User.Identity.Name!
 </li>
 <li>
 @Html.ActionLink("Sign out", "SignOut",
 "Account")
 </li>
 </ul>
 </text>
}
else
{
 <ul class="nav navbar-nav navbar-
 right">
 <li>@Html.ActionLink("Sign in",
 "SignIn", "Account", routeValues: null,
 htmlAttributes: new { id = "loginLink"
 })</li>
 </ul>
}
```

Here in this file, we are simply creating a SignIn and SignOut button and telling the application to invoke Account controller action on SignIn button click. We are also adding the name of the user at runtime through the code User.Identity.Name, to welcome him/her!

Now we need to link this _loginPartial.cshtml on layout page as highlighted below, therefore change the code of Layout.cshtml to the following code –

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<meta name="viewport"
content="width=device-width, initial-
scale=1.0">
<title>@ViewBag.Title - My ASP.NET
Application</title>
@Styles.Render("~/Content/css")
@Scripts.Render("~/bundles/modernizr")
</head>
<body>
 <div class="navbar navbar-inverse
 navbar-fixed-top">
  <div class="container">
  <div class="navbar-header">
   <button type="button" class="navbar-
   toggle" data-toggle="collapse" data-
   target=".navbar-collapse">
   <span class="icon-bar"></span>
   <span class="icon-bar"></span>
   <span class="icon-bar"></span>
   </button>
   @Html.ActionLink("Application name",
  "Index", "Home", null, new { @class =
  "navbar-brand" })
  </div>
  <div class="navbar-collapse collapse">
   <ul class="nav navbar-nav">
   <li>@Html.ActionLink("Home", "Index",
   "Home")</li>
   <li>@Html.ActionLink("About",
   "About", "Home")</li>
   <li>@Html.ActionLink("Contact",
   "Contact", "Home")</li>
   </ul>
   @Html.Partial("_LoginPartial")
  </div>
 </div>
</div>
<div class="container body-content">
 @RenderBody()
 <hr />
 <footer>
  <p>&copy; @DateTime.Now.Year - My ASP.
  NET Application</p>
 </footer>
</div>
@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/bootstrap")
@RenderSection("scripts", required:
false)
</body>
</html>
```

If we refer to the _LoginPartial.cshtml code, you will see that we are referring to the controller name as Account for sign in and out functionality. Therefore let's add AccountController in controller's folder. Right click on Controllers folder > Add > Controller > MVC 5 Controller – Empty > name as AccountController. Replace the following code –

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

// The following using statements were
added for this sample.
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.
WsFederation;
using Microsoft.Owin.Security;

namespace DNCMvcApp.Controllers
{
 public class AccountController :
 Controller
 {
  public void SignIn()
  {
   // Send a WSFederation sign-in
   request.
   if (!Request.IsAuthenticated)
   {
    HttpContext.GetOwinContext().
    Authentication.Challenge(new
    AuthenticationProperties
    { RedirectUri = "/" },
    WsFederationAuthenticationDefaults.
    AuthenticationType);
   }
  }
  public void SignOut()
  {
   // Send a WSFederation sign-out
   request.
   HttpContext.GetOwinContext().
   Authentication.SignOut(
   WsFederationAuthenticationDefaults.
   AuthenticationType,
   CookieAuthenticationDefaults.
   AuthenticationType);
  }
 }
}
```

If you observe the code, we are invoking the Challenge() method, and we are checking the return code. If it is 401, it means unauthorized, so we are asking the user to login again.

Web Config changes

Registration application is already done in Azure Ad. Now we need to make changes in our application to make our application know about Azure AD. So add the following configuration settings in web.config file –

```
<add key="ida:Wtrealm" value="[Enter the
App ID URI]" />
<add key="ida:AADInstance"
value="https://login.microsoftonline.
com" />
<add key="ida:Tenant" value="[Enter
Azure AD tenant name]" />
```

If we replace the values in the above configuration settings with our values, then it would be as follows –

```
<add key="ida:Wtrealm" value="http://
dotnetcurryad/wsfedsample_owin" />
<add key="ida:AADInstance"
value="https://login.microsoftonline.
com" />
<add key="ida:Tenant"
value="dotnetcurryad.onmicrosoft.com" />
```

We are done with required application changes and all set to run it. Press F5 to run the application in debug mode. Click on Sign In button. Now let's go back to understand more about users in Azure active directory.

Add School or Work account User to Azure AD tenant

Microsoft always reiterates that they are making life of developers simple, but as per my experience, sometimes things are not so simple with Microsoft (pun intended). It starts with your account itself. First we need to understand the different types of accounts Azure AD supports. As of today you have 2 accounts –

- Microsoft Account (previously Live ID)
- Work or school account (previously Organizational account)

It sometimes becomes hard to keep up with frequent name changes. For example when Azure was introduced it was "Windows Azure" now everyone talks about "Microsoft Azure". Similarly Live Id is now called as Microsoft account and Organizational account is now called "Work or School account".

So if you create a new user in Azure AD, then it becomes "Work or School account" or "Azure AD based account".

When we created an Azure AD tenant, the current logged in user of the Azure Portal automatically became the global admin of the tenant. The current logged in user in Azure portal I have used is "Microsoft Account and not the Azure AD account".

For signing in to the DNCMvcApp we can definitely use a Microsoft account user, but this is NOT CORRECT.

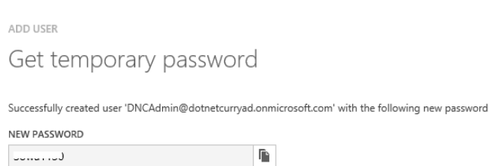
Microsoft Account can be used with Azure Active Directory when they are used with Microsoft Applications, like the Azure Portal. Microsoft Accounts are not expected to work with applications registered by developers YET. Unfortunately if you use it today, it may work because it is a bug, which hopefully will be rectified soon.

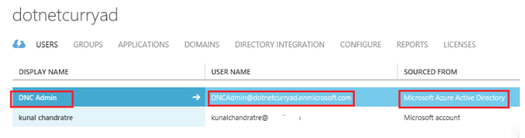
So *ideally* only Azure AD based user should be allowed to login in DNCMvcApp application. So let's create a user in Azure AD tenant.

Click on Add User button in Azure AD tenant page in Azure Management Portal. Provide the values as shown here –

Type of user	New user in your organization
User Name	DNCAdmin
Tenant to be selected	Dotnetcurryad.onmicrosoft.com
First Name	DNC
Last Name	Admin
Display Name	DNC Admin
Role	Global Administrator
Enable Multi Factor authentication	Keep it unchecked.
Alternate Email Address	<Email of your choice>

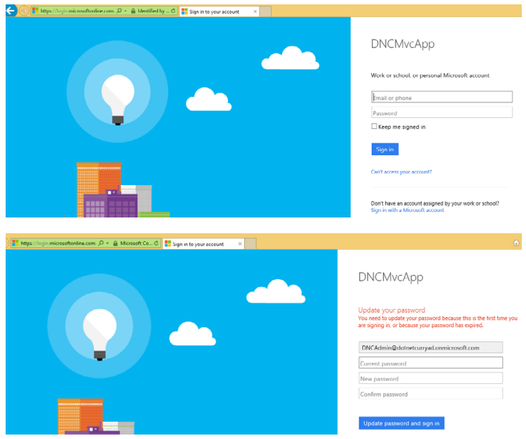
In the last screen, a prompt will appear to "Create a temporary password". Click on Create button and record this password for future use. When you login to DNCMvcApp using this user, you will need to change this temporary password to a new one.



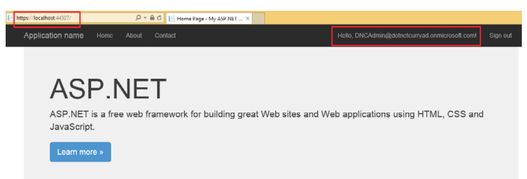


So this user DNCAdmin@dotnetcurryad.onmicrosoft.com now we will be used for logging in to the DNCMvcApp application.

Run the application and click on SignIn button. Once prompted, put the DNCAdmin credentials. You will prompted immediately to change the temporary password as shown here –



Once you update the password, you will be logged in to DNCMvcApp application and username will appear in right most corner as shown here –



Secure ASP.NET MVC application using Azure AD and WIF

In order to secure our MVC application using Azure AD and Windows Identity Foundation (WIF), please refer to an article written by Mahesh Sabnis - http://www.dotnetcurry.com/windows-azure/1123/secure-aspnet-mvc-azure-using-active-directory-signon.

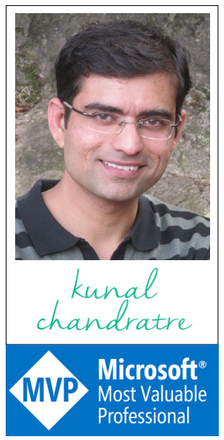
That's all. As we just saw, it is relatively quite easy to secure an ASP.NET MVC application using Azure Active Directory tenant either using WIF or OWIN with Ws-Federation.

Conclusion

In this article, we saw some important and essential concepts around Azure Active Directory. We also explored the structure of Azure AD, and why it is commonly referred to as multi-tenant service. We also saw how an ASP.NET MVC application can be secured with Azure Active Directory using OWIN and WIF with WS-Federation protocol.

Download the entire source code from GitHub at bit.ly/dncm23-azure-active-dir

About the Author



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

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

- ASP.NET
- MVC, WEB API
- ANGULAR.JS
- NODE.JS
- AZURE
- VISUAL STUDIO
- .NET
- C#, WPF

We've got it all!

85K PLUS READERS

200 PLUS AWESOME ARTICLES

22 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

THANK YOU

FOR THE 23rd EDITION


@yacoubmassad


@kunalchandratre


@sravi_kiran


@craigber


@maheshdotnet


@shobankr


@damirarh


@suprotimagarwal


@saffronstroke

WRITE FOR US