ISSUE 24 MAY-JUN 2016

DNCMagazine

www.dotnetcurry.com

C# 7.0
What to expect?

TypeScript
Overview

Clean
Composition
Roots
with Pure DI

C# Quiz

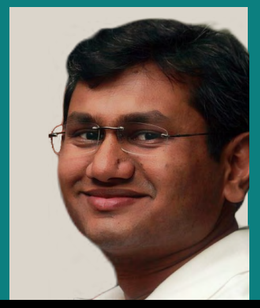
Issue 24

Microsoft Bot
Framework
and Cognitive Services

SOLID PRINCIPLES
Dependency Inversion

Migrating from SQL Server to
NoSQL

CONTENTS



Editor in Chief

There's never a dull day for a developer keeping up with the Microsoft stack. The last couple of weeks caused a bit of a stir with the annoucement of SQL Server on Linux (yes, read that again), Windows 10 supporting Bash, Xamarin acquisition, Project Continuum, Centennial, Bot Framework, Cognitive Services and the plethora of other tools and technologies announced at Build 2016. Each of these annoucements laid out a new and intriguing path for developers to connect with the Microsoft Developer Ecosystem.

With Microsoft focusing on cross-platform development with its own open source version of .NET Core; C# is back into the mainstream. This month's edition focuses on the next version of C#, SOLID principles and DI. There's a demo on the latest Bot Framework and an article on using Azure DocumentDB to migrate to NoSQL. For our JavaScript devs, we have a nice overview on TypeScript.

So what's next? Are pigs gonna sprout wings now? Well time will tell, but for now, it is time for developers to rejoice and reinvent themselves!

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

Suprotim Agarwal

CREDITS

Editor In Chief
Suprotim Agarwal
suprotimagarwal@a2zknowledgevisuals.com

Art Director
Minal Agarwal
minalagarwal@a2zknowledgevisuals.com

Contributing Authors
Craig Berntson
Damir Arh
Mahesh Sabnis
Ravi Kiran
Shoban Kumar
Suprotim Agarwal
Yacoub Massad

Technical Reviewers
Damir Arh
Suprotim Agarwal
Yacoub Massad

Next Edition
5th July 2016

www.dotnetcurry.com/magazine

POWERED BY

a2z | Knowledge Visuals

dot net curry.com
come to where the .NET flavor is

What Features Can We Expect in C# 7.0

The current incarnation of C# compiler (better known as Roslyn) was open sourced in April 2014. Not only is the compiler now being developed on GitHub; the language design is also openly discussed in public. This allows all interested parties to see how the next version of the language might look like. This article provides an overview of the current state of thought process involved while designing new features. If you are interested in a broader aspect of the current Roslyn ecosystem, you can read my article in the March 2016 edition of DotNetCurry (DNC) magazine: .NET Compiler Platform (a.k.a. Roslyn) – An Overview [bitly.com/dnc-roslyn-overview]

Main Theme for the Next Release of C#

New features in each version of C# so far (with the exception of C# 6.0 maybe) have revolved around a specific theme:

- C# 2.0 introduced generics.
- C# 3.0 enabled LINQ by bringing extension methods, lambda expressions, anonymous types and other related features.
- C# 4.0 was all about interoperability with dynamic non-strongly typed languages.
- C# 5.0 simplified asynchronous programming with the `async` and `await` keywords.
- C# 6.0 had its compiler completely rewritten from scratch, and introduced a variety of small features and improvements that were easier to implement now. You can find an overview of all new C# 6.0 features in the Jan 2016 edition of the DotNetCurry (DNC) magazine: Upgrading Existing C# Code to C# 6.0. [bitly.com/dnc-upgrading-csharp]

C# 7.0 will probably be no exception to this rule. The language designers are currently focusing on three main themes, which are as follows:

- **Increased usage of web services** is changing the way data is being modelled. Instead of designing the data models as a part of the application, their definition is becoming a part of web service contracts. While this is very convenient in functional languages, it can bring additional complexity to object oriented development. **Several C# 7.0 features are targeted at making it easier to work with external data contracts**.
- **Reliability and robustness** is a constant challenge in software development. C# 7.0 might contribute a portion of its development time to address this challenge.
- **Increased share of mobile devices** is making performance an important consideration again. There are planned features for C# 7.0 that could allow **performance optimizations**, which were previously not possible in the .NET framework.

Let us take a closer look at some of the planned features belonging to each of these themes.

Working with Data

Object oriented languages like C# work great in scenarios with a predefined set of operations acting on an extensible set of data types. These are typically being modelled with an interface (or a base class) specifying the available operations and a potentially growing number of classes representing the data types. By implementing the interface, classes contain the implementation of all the expected operations.

For example, in a game, the classes could be different types of weapons (such as a sword or a bow), and the operations could be different actions (such as attack or repair). In this scenario, adding a new weapon type (e.g. a lightsaber) would be simple: just create a new class that implements the weapon interface. Adding a new action (e.g. turn) on the other hand would require extending the interface and modifying all the existing weapon implementations. This approach feels very natural in C#.

```
interface IEnemy
{
  int Health { get; set; }
}

interface IWeapon
{
  int Damage { get; set; }
  void Attack(IEnemy enemy);
  void Repair();
}

class Sword : IWeapon
{
  public int Damage { get; set; }
  public int Durability { get; set; }

  public void Attack(IEnemy enemy)
  {
    if (Durability > 0)
    {
      enemy.Health -= Damage;
      Durability--;
    }
  }

  public void Repair()
  {
    Durability += 100;
  }
}

class Bow : IWeapon
{
  public int Damage { get; set; }
  public int Arrows { get; set; }

  public void Attack(IEnemy enemy)
  {
    if (Arrows > 0)
    {
      enemy.Health -= Damage;
      Arrows--;
    }
  }

  public void Repair()
  { }
}
```

In functional programming, data types do not include operations. Instead, ***each function implements a single operation for all the data types***. This makes it much easier to add a new operation (just define and implement a new function), but much more difficult to add a new data type (modify all existing

functions accordingly). While this is already possible in C#, it is much more verbose than it could be.

```
interface IEnemy
{
  int Health { get; set; }
}

interface IWeapon
{
  int Damage { get; set; }
}

class Sword : IWeapon
{
  public int Damage { get; set; }
  public int Durability { get; set; }
}

class Bow : IWeapon
{
  public int Damage { get; set; }
  public int Arrows { get; set; }
}

static class WeaponOperation {
  static void Attack(this IWeapon
  weapon, IEnemy enemy)
  {
    if (weapon is Sword)
    {
      var sword = weapon as Sword;
      if (sword.Durability > 0)
      {
        enemy.Health -= sword.Damage;
        sword.Durability--;
      }
    }
    else if (weapon is Bow)
    {
      var bow = weapon as Bow;
      if (bow.Arrows > 0)
      {
        enemy.Health -= bow.Damage;
        bow.Arrows--;
      }
    }
  }

  static void Repair(this IWeapon
  weapon) {
    if (weapon is Sword)
    {
      var sword = weapon as Sword;
      sword.Durability += 100;
    }
  }
```

Pattern matching is the feature that should help simplify the above code. Let us apply it to the `Attack` method in a step-by-step manner:

```
static void Attack(this IWeapon weapon,
IEnemy enemy)
{
  if (weapon is Sword sword)
  {
    if (sword.Durability > 0)
    {
      enemy.Health -= sword.Damage;
      sword.Durability--;
    }
  }
  else if (weapon is Bow bow)
  {
    if (bow.Arrows > 0)
    {
      enemy.Health -= bow.Damage;
      bow.Arrows--;
    }
  }
}
```

Instead of checking the type of weapon and assigning it to a correctly typed variable in two separate statements, the `is` operator will now also allow us to declare a new variable and assign the type cast value to it.

With a similar result, a `switch case` statement can be used instead of `if`. This can make the code even more clear, especially when there are many different branches:

```
switch (weapon)
{
  case Sword sword when sword.Durability
  > 0:
    enemy.Health -= sword.Damage;
    sword.Durability--;
    break;
  case Bow bow when bow.Arrows > 0:
    enemy.Health -= bow.Damage;
    bow.Arrows--;
    break;
}
```

Notice, how the *case* statement performs both the type cast assignment, and the additional conditional check, increasing the terseness of the code.

Another pattern matching related feature is **switch expressions**. You can think of them as `switch` statements returning a value from each `case` branch. Using this feature, transitions for a simple finite state machine could be defined in a single expression bodied method:

```
static State Request(this State state,
Transition transition) =>
 (state, transition) match
 (
   case (State.Running, Transition.
   Suspend): State.Suspended
   case (State.Suspended, Transition.
   Resume): State.Running
   case (State.Suspended, Transition.
   Terminate): State.NotRunning
   case (State.NotRunning, Transition.
   Activate): State.Running
   case *: throw new
   InvalidOperationException()
 );
```

The above code takes advantage of another new feature: **tuples**. They are supposed to be an even more lightweight alternative to anonymous classes. Their primary use will probably be in functions that return multiple values, as an alternative to `out` arguments:

```
public (int weight, int count)
Stocktake(IEnumerable<IWeapon> weapons)
{
  var w = 0;
  var c = 0;
  foreach (var weapon in weapons)
  {
    w += weapon.Weight;
    c++;
  }
  return (w, c);
}
```

A more functional approach to development will quickly result in classes that act as value containers only, and do not include any methods or business logic. The **records** syntax should allow standardized implementation of such classes with absolute minimum code:

```
public class Sword(int Damage, int
Durability);
```

This single line will result in a fully functional class:

```
public class Sword : IEquatable<Sword> {
 public int Damage { get; }
 public int Durability { get; }

 public Sword(int Damage, int
 Durability) {
  this.Damage = Damage;
  this.Durability = Durability;
 }

 public bool Equals(Sword other) {
  return Equals(Damage, other.Damage) &&
  Equals(Durability, other.Durability);
 }

 public override bool Equals(object
 other) {
  return (other as Sword)?.Equals(this)
  == true;
 }

 public override int GetHashCode() {
  return (Damage.GetHashCode() * 17 +
  Durability.GetHashCode())
  .GetValueOrDefault();
 }

 public static void operator is(Sword
 self, out int Damage, out int
 Durability) {
  Damage = self.Damage;
  Durability = self.Durability;
 }

 public Sword With(int Damage =
 this.Damage, int Durability = this.
 Durability) =>
 new Sword(Damage, Durability);
}
```

As you can see, the class features *read-only* properties, and a constructor for initializing them. It implements value equality and overrides `GetHashCode` correctly for use in hash-based collections, such as `Dictionary` and `Hashtable`. You probably do not recognize the last two functions:

- `Is` operator overloading allows decomposition into a tuple in pattern matching constructs.
- For the explanation of `With` method, read the following section.

Records will support inheritance, but the exact syntax is not decided, yet.

Increased Reliability

The `Sword` class generated from the record syntax that we just saw, is an example of an **immutable class**. This means that its state (the value of its properties) cannot be changed after an instance of the class is created.

If you are wondering how that relates to reliability, think multithreaded programming. With processors gaining additional cores instead of higher clock speeds, mutithreaded programming is only going to become more important and prevalent in server, desktop, and mobile applications. While immutable objects require a different approach to programming, they by design prevent race conditions caused by multiple threads modifying the same object without proper synchronization (i.e. without the correct use of `lock` or other thread synchronization primitives).

Although it is currently possible to create immutable objects in C#, it is more complicated than it needs to be. The following changes in C# 7.0 should make it more convenient to define and work with immutable objects:

- Object initializers will work for read-only properties, automatically falling back to a matching constructor:

```
IWeapon sword = new Sword { Damage = 5,
Durability = 500 };
```

- A special syntax will be available for concise creation of modified copies of objects:

```
IWeapon strongerSword = sword with {
Damage = 8 };
```

The `with` expression above will create a copy of the `sword` object, with all properties having the same value, except `Damage` being set to the newly provided value. Complete details on the inner workings of this expression is still under discussion. One of the options is requiring the classes to have a `With` method that will be used, as presented in the records example:

```
public Sword With(int Damage = this.
Damage, int Durability = this.
Durability) =>
 new Sword(Damage, Durability);
```

This would allow the `with` expression syntax to be automatically converted into the following method call:

```
IWeapon strongerSword = sword.
With(Damage: 8);
```

The second part of reliability efforts in C# 7.0 is the subject of **null safety**. We can probably all agree that `NullReferenceException` is one of the most common and most difficult to troubleshoot failure conditions. Any language improvement that could reduce the number of such exceptions would certainly have a positive impact on the overall application reliability.

Third party vendors, such as JetBrains with their renowned ReSharper extension for Visual Studio have already taken first steps in this direction. Their work is based on static analysis of code, issuing warnings whenever the developer tries to dereference an object which has not been checked for null value before. This is complemented by attributes that can be used to annotate the methods with the information whether they can return a null value or not. They also prepared the annotations for the .NET BCL (base class library) classes. If a developer would correctly annotate all of his/her code, static analysis should be able to warn reliably about any potential `NullReferenceException` sources.

The C# language design team is trying to achieve the same goal, albeit at the language level. The core idea is to allow variable type definitions to contain information whether they can have a null assigned to them or not:

```
IWeapon? canBeNull;
IWeapon cantBeNull;
```

Assigning a null value or a potential null value to a non-nullable variable would result in a compiler warning (the developer could configure the build to fail in case of such warnings, to be extra safe):

```
canBeNull = null;       // no warning
cantBeNull = null;      // warning
cantBeNull = canBeNull; // warning
```

The problem with such a change is that it breaks existing code: it is assumed that all variables from before the change are non-nullable. To cope with that, static analysis for null safety can be disabled at the project level. The developer can decide when he/she wants to opt-in for nullability checking.

Similar changes to C# have already been considered in the past, but never came to fruition because of the backward compatibility issues. Since Roslyn has considerably changed what the compiler and the diagnostics performing the static analysis are capable of, the language team has decided to revisit the subject once more. Let us keep our fingers crossed that they manage to come up with a feasible solution.

Improved Performance

The performance improvements in C# 7.0 focuses on reducing the copying of data between memory locations.

Local functions will allow declaration of helper functions nested inside other functions. This will not only reduce their scope, but also allow the use of variables declared in their encompassing scope, without allocating additional memory on heap or stack:

```
static void ReduceMagicalEffects(this
IWeapon weapon, int timePassed) {
  double decayRate =
  CalculateDecayRate();
  double GetRemainingEffect(double
  currentEffect) =>
  currentEffect * Math.Pow(decayRate,
  timePassed);

  weapon.FireEffect =
  GetRemainingEffect(weapon.FireEffect);
  weapon.IceEffect =
  GetRemainingEffect(weapon.IceEffect);
  weapon.LightningEffect =
  GetRemainingEffect(weapon.
  LightningEffect);
}
```

Return values and local variables by reference can also be used to prevent unnecessary copying of data. At the same time, they modify the behavior. Since the variable is pointing at the original memory location, any changes to the value there will of course also affect the local variable value:

```
[Test]
public void LocalVariableByReference()
{
  var terrain = Terrain.Get();

  ref TerrainType terrainType = ref
  terrain.GetAt(4, 2);
  Assert.AreEqual(TerrainType.Grass,
  terrainType);

  // modify enum value at the original
  location
  terrain.BurnAt(4, 2);
  // local value was also affected
  Assert.AreEqual(TerrainType.Dirt,
  terrainType);
}
```

In the above example, `terrainType` is a local variable by reference, and `GetAt` is a function returning a value by reference:

```
public ref TerrainType GetAt(int x, int
y) => ref terrain[x, y];
```

Slices is the final performance related proposed feature:

```
var array = new int[] { 1, 2, 3, 4, 5 };
var slice = Array.Slice(array, 0, 3); //
refers to 1, 2, 3 in the above array
```

Slices would make it possible to treat parts of an existing array as a separate array, while still pointing at the same memory location within the original location.

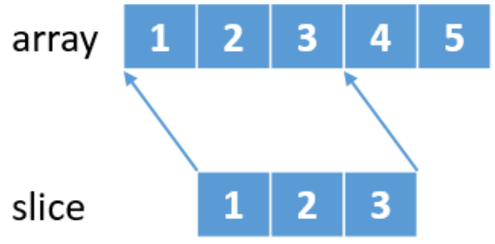


Image 1: Slices are parts of another array

Again, any modification to one of the arrays would affect both at the same time, without any values being copied. This could result in a much more effective management of large states, e.g. in games. All the required memory could be allocated only once at the beginning of the application (or level), completely avoiding new memory allocations and garbage collection.

As a step further, it could be possible to get a chunk of natively allocated memory in a similar manner, reading from and writing to it directly without marshalling.

Trying Out the Experimental Features

Although all of the above features are far from complete, any work already done on them is already available on GitHub. If you are interested in trying it out, you can do so.

As of this writing, the simplest way is to install the Visual Studio "15" Preview that is available for download since the end of March. It includes a new C# compiler with the following experimental features waiting for you to try: **pattern matching, local functions,** and **return values and local variables by reference**.

The less mature features would require building your own version of the compiler from the GitHub sources, which is beyond the scope of this article. You can find articles with detailed instructions [bit.ly/dnc-own-compiler] if you are interested.

Even in Visual Studio "15" Preview, the new experimental language features will not be available to you by default.

```
Feature 'pattern matching' is experimental and unsupported; use '/features:patterns' to enable.
```

Image 2: Experimental features must be enabled explicitly

In spite of the instructions in the error, the simplest way to enable these features at the time of this writing is to add `__DEMO__` and `__DEMO_EXPERIMENTAL__` conditional compilation symbols in the build properties of your project.

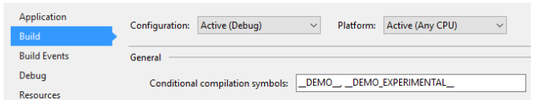


Image 3: Add conditional compilation symbol

Now you should be able to use any currently supported experimental language features, and compile the project without errors.

Conclusion:

All the new language features described in this article are still work-in-progress. In the final version of C# 7.0, they might be significantly different or not available at all. This article serves only as an overview of the current state of the C# Language so to give you a glimpse into the future, and maybe spark enough interest to make you follow the development more closely, or even try some of the features out, before they are done. By taking a more active part in the language development, you can influence it, and at the same time learn something new; potentially improving your existing coding practices even before the next version of the language is finally available ■

About the Author

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.

Clean Composition Roots with Pure DI

When we apply the SOLID principles, the result is usually a lot of small classes, each having a single responsibility, along with interfaces that have a number of smaller methods.

Dependency Injection (DI) is the preferred way of making these classes interact with each other. Classes declare their dependencies in their constructors, and a separate entity is responsible for providing these dependencies. This entity is the Composition Root.

The Composition Root is the place where we create and compose the objects resulting in an object-graph that constitutes the application. This place should be as close as possible to the entry point of the application.

When we apply the Open/Closed Principle (the O in SOLID) [bitly.com/dnc-open-closed], the normal change process during maintenance involves creating a new class or classes that implement the new or modified system behavior. We then go to the Composition Root and make sure that an instance of this class or these classes are injected in the appropriate location in the object graph. This process should involve no modification of existing classes (other than the Composition Root). This process makes SOLID code append-only.

DI containers are tools that intend to help developers compose the object graph easily. However, the usage of such tools is optional. Pure DI is the application of Dependency Injection without using a DI container. When we use Pure DI, we construct the objects manually and inject their dependencies manually.

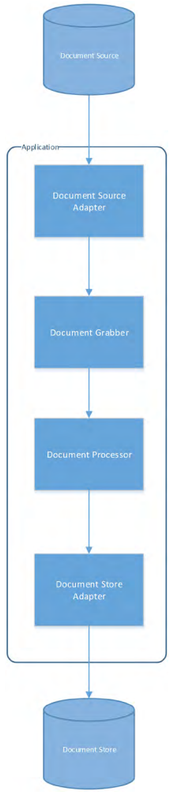
It's important to make sure that the Composition Root is easy to read and navigate, so that the change process is done in an easy and quick manner.

In this article, I am going to show how we can use Pure DI and clean code practices to create large Composition Roots that are readable and maintainable. I will show how a normal change process would look like.

An example: initial build

Let's say that we are building a document indexing application. This application is required to collect documents from some source, store them in a database, and index them so that other applications can allow users to search the documents in this database.

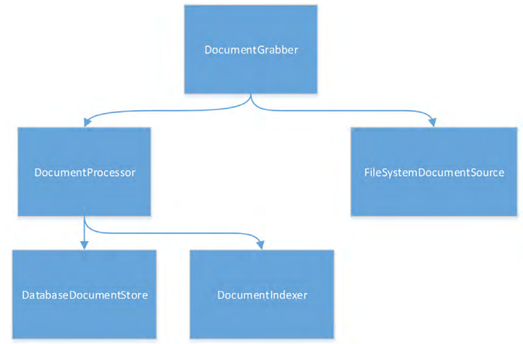
A high level design of such an application is expected to look like the following:



Basically, the **Document Grabber** component pulls documents from some source. It then gives them to the **Document Processor** component, which in turn processes them and sends them to the **Document Store**.

In this article, I am going to use the term *component* to refer to a group of classes (or even a single class) in the object graph that is responsible for some function in the system.

So we start building the application, and a set of classes and interfaces emerge. The following figure shows some of the classes and how they are supposed to be wired together in the Composition Root:



Basically, the `DocumentGrabber` class requires an `IDocumentSource` object to pull documents from, and an `IDocumentProcessor` object to which it will give the documents for processing. When we compose the application, we use the `FileSystemDocumentSource` class as our document source. This is because the system is required to pull documents from the file system only. As a document processor, we use the `DocumentProcessor` class. This class uses a `DocumentIndexer` to extract information about the document (e.g. words in the document). It also depends on `IDocumentStore` which is used to store documents with their corresponding indexing information.

Now, here is how we can use Pure DI to wire these objects:

```
static void Main(string[] args)
{
  //...
  var documentGrabber =
  new DocumentGrabber(
    new FileSystemDocumentSource(path),
    new DocumentProcessor(

      new DocumentIndexer(),
      new DatabaseDocumentStore
      (storeConnectionString)));
      //...
}
```

The variables 'path' and 'storeConnectionString' are settings that come from the configuration file.

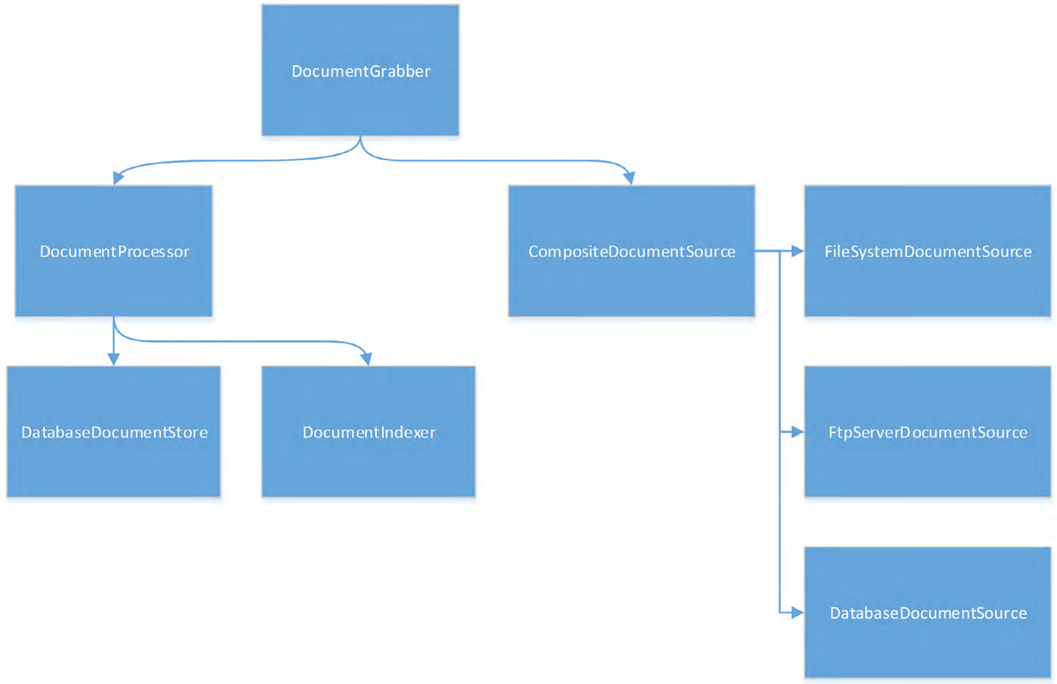
Currently, this Composition Root is very small. We simply have it in a single method.

New requirements

Now, we get a new requirement to add support for pulling documents from an FTP server and from some database that belongs to another application. Here is how such a change process looks like:

We first create new implementations of the `IDocumentSource` interface to support this new behavior. Namely, we create the `FtpServerDocumentSource` and `DatabaseDocumentSource` classes.

Secondly, since the `DocumentGrabber` class depends on a single `IDocumentSource`, we need to create an implementation of such an interface that allows us to collect documents from multiple sources. We use the composite design pattern to create a `CompositeDocumentSource`. This class takes an array of `IDocumentSource` objects in the constructor and implements the `IDocumentSource` interface by pulling documents from all the injected document sources. Here is how the application graph looks like now (see figure below):



Finally, we create instances of the new classes in the Composition Root and then compose them using Pure DI like this:

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

 var documentGrabber =
 new DocumentGrabber(
 new CompositeDocumentSource(
 new FileSystemDocumentSource(path),
 new DatabaseDocumentSource
 (otherApplicationConnectionString),
 new FtpServerDocumentSource
 (ftpServerAddress)),
 new DocumentProcessor(
 new DocumentIndexer(),
 new DatabaseDocumentStore
 (storeConnectionString)));

 //...
}
```

Now, the Composition Root is getting bigger. Let's refactor it. One advice from the Clean Code book by Robert C. Martin is that ***functions should have a single level of abstraction***. We can apply this principle for object composition. The main method should still create the `DocumentGrabber`, but let's extract the code that creates the document source sub-graph, into a separate method. Here is how our code looks like now:

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

 var documentGrabber =
 new DocumentGrabber(
 CreateDocumentSource(),
 new DocumentProcessor(
 new DocumentIndexer(),
 new DatabaseDocumentStore
 (storeConnectionString)));

 //...
}

static IDocumentSource
CreateDocumentSource()
{
 //...

 return new CompositeDocumentSource(
 new FileSystemDocumentSource(path),
 new DatabaseDocumentSource
 (otherApplicationConnectionString),
 new FtpServerDocumentSource
 (ftpServerAddress));
}
```

Now, the `CreateDocumentSource` method is responsible for creating the document sources and composing them together.

Yet another requirement

The system has received yet another requirement. The system needs to support any number of document sources. The administrator should be able to define a list of document sources inside some XML configuration file. What should we do next?

The first thing we do is visit the Composition Root. This is because the Composition Root constitutes the application. We should be able to understand the structure of the application from there.

So, we go to the Composition Root, and we take a look at the main method. From there it is easy to see that the `CreateDocumentSource` method should be our next place to visit. We go to such a method and find that we explicitly create 3 document sources. What we need to do next is to change the configuration system (which I am not showing) to support specifying a list of document sources. Here is how a fragment of the configuration file would look like:

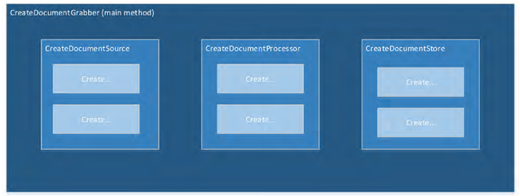
```
<DocumentSources>
 <DocumentSource Type="FileSystem"
 Path="c:\\test"/>
 <DocumentSource Type="FileSystem"
 Path="c:\\test2"/>
 <DocumentSource Type="FtpServer"
 Address="ftpserver1"/>
 <DocumentSource Type="Database"
 ConnectionString="Server=..."/>
</DocumentSources>
```

Now, assuming that our configuration system has already been modified, we modify the `CreateDocumentSource` method like this:

```
static IDocumentSource
CreateDocumentSource ()
{
 List<SourceSettings>
 sourceSettingEntries =
 GetSourceSettingEntries();

 return new CompositeDocumentSource(
  sourceSettingEntries
  .Select(entry =>

  CreateDocumentSourceFromSettingsEntry
  (entry))
  .ToArray());
}

static IDocumentSource
CreateDocumentSourceFromSettingsEntry
(SourceSettings entry)
{
  if(entry.Type ==
  DocumentSourceType.FileSystem)
  return new FileSystemDocumentSource
  (entry.Path);

  if (entry.Type ==
  DocumentSourceType.FtpServer)
  return new FtpServerDocumentSource
  (entry.Address);

  if (entry.Type ==
  DocumentSourceType.Database)
  return new FileSystemDocumentSource
  (entry.ConnectionString);

  throw new Exception("Invalid source
  type");
}
```

The examples I gave are simple examples. But the idea here is that the main method of the Composition Root should only contain code that creates the high-level components of the application. This code is usually method call invocations to create components. Then, inside each of these methods, you find code at a lower level of abstraction. For example, for the document source component, you find code that creates individual classes that act as document sources.

As you saw, the complexity of creating the document source component increased with time. As the complexity increases, we extract more methods so that code in each method is at the same level of abstraction. If we follow this rule, then the Composition Root would look like this:



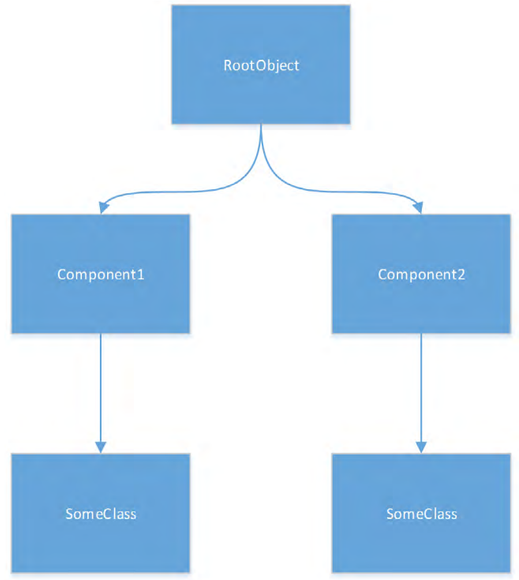
Basically, the Composition Root becomes a tree of components. At the root of the tree, you can see the high-level components of the system. As you go a single level down the tree, you see the sub-components of the components in the level above. In this way, the Composition Root *constitutes* the application. If you need to understand the structure of the application on different levels, you can go to the Composition Root.

The tree-like structure of the Composition Root makes it *navigatable*. When you are modifying the application, you can easily navigate from the main method to the method that is responsible for creating the sub-object-graph that you need to modify. For instance, you look at the main method and you find that it calls three methods; `CreateDocumentSource`, `CreateDocumentProcessor`, and `CreateDocumentStore`. These methods create the high-level components of the system. From there, you can pick the component under which the new/modified behavior should be. You then go to the corresponding method to see what sub components are there and do the same thing. You should always give names to these methods that makes them easy to understand and thus makes the navigation process easier.

The tree-like structure for Composition Roots is great. We should always try to make the Composition Root structure that way. However, this is not always possible. The main reason for this is shared components.

Code sharing versus Resource/state sharing

We should always try to make the object graph that we create, tree-like. If two components of the system depend on the same class, we should prefer to give them two different instances of the class. This allows us to make the graph tree-like. This is only possible if this class does not hold resources that the two components would like to share. I call this *code sharing*. Consider the following figure:

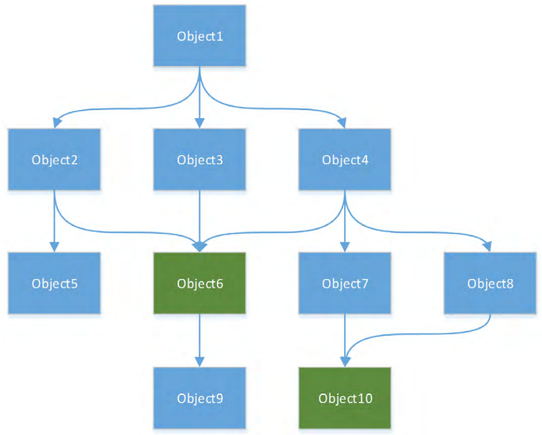


In this figure, two components are using two different instances of *SomeClass* class. They share the code of this class, but they don't share any single resource. SomeClass could have some state, but component1 and component2 don't need to share it, i.e., each component has its own copy of the state.

Resource sharing on the other hand is when you want to have two or more components use the same instance of a class. This is needed in some cases where that class holds a shared resource like shared state. For example, consider a system where you want some components to temporarily pause the whole system in case of an error. You would need a class or a group of classes to manage such pause/resume process. Particularly, some class needs to hold the state of whether the system is currently paused. An instance of such a class needs to be shared across all components that need to pause/resume the system or query the state of the system.

How to deal with resource sharing

In cases where it is required to share an instance of a class between components, we should make sure that we create such an instance at the latest possible point. Consider the following object graph as an example:



In this graph, green objects are instances that are shared between multiple objects. These are the objects that make the graph *not* tree-like. What we can do is to create object 6 and object 10 in the main method, and then pass them down the methods that create the different components. While this works, it will force us to make many methods on the way down, have parameters to deliver shared dependencies to components. These parameters might not make sense at the upper levels, but we are forced to have them.

To mitigate this issue, we can only create these objects (6 and 10) at the point when they are needed. For object 6, we need to do it before we create object1 because we have to inject it into objects 2, 3, and 4 which are direct dependencies of object 1. However, for object 10, we can move its creation to the method that creates object 4 because it is only needed for objects that are direct dependencies of object 4. Here is how the Composition Root would look like in this case:

```
static void Main(string[] args)
{
  var object6 = new Object6(
  new Object9());

  var object1 =
  new Object1(
    CreateObject2(object6),
    CreateObject3(object6),
    CreateObject4(object6));
}

static Object4 CreateObject4(Object6
object6)
{
  var object10 = new Object10();

  return new Object4(
    object6,
    new Object7(
         object10),
        new Object8(
            object10));
}
```

The creation of object6 in our case is done in the same method that creates object1. Although this might violate the rule of a single level of abstraction per function, we are forced to do it.

Please note that object10 is not created in the main method but in the CreateObject4 method. This can be done since object10 is not needed outside the scope of the sub-object-graph that starts at object4.

Shared dependencies as private fields in the Composition Root

Imagine in the last example that the same instance of object10 that is used by object7 and object8, is also required by object5 and object6. This means that we have to create object10 in the main method and pass it multiple levels down.

One approach to solve this issue is to create object10 in the main method, but save it as a private field. This way, methods at the lower levels can access it without having many methods pass such a dependency. However, at some point, when you have multiple shared instances of the same type, it might become hard to manage them and reason about them.

DI containers

DI containers are tools that aim to help developers create Composition Roots. In my experience, when used with big applications, they actually make the Composition Root less readable and maintainable. For more information about this point of view, you can read my article bitly.com/why-di-containers-fail.

Summary

In this article, I have shown how we can use Pure DI and the single level of abstraction per function rule to create Composition Roots that we can understand and navigate easily. This will help a lot with the maintenance process. A normal change process involves creating new classes for the new or modified behavior, and then going to the now easy-to-navigate Composition Root and make sure that we inject instances of the new classes into the appropriate location.

We should always try to make the structure of the Composition Root, tree-like. This helps a lot with keeping each method concerned with a single level of abstraction. We should prefer to give different components different instances of dependencies, as long as they don't require resource sharing ■

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 criticalsoftwareblog.com.

MICROSOFT BOT FRAMEWORK & COGNITIVE SERVICES

One of the main reasons I love programming is you continue learning new and exciting things. Your job is never monotonous if you are doing what you love! Big software companies like Microsoft, Google, Facebook and many more are all trying to put Innovation at the forefront and working on releasing interesting products, frameworks and APIs.

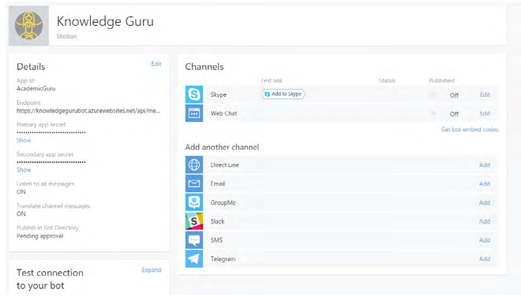
As a Microsoft developer, I find the Build conference to be one of the most exciting places to learn about these innovations. This year's Build held in March 2016 was even better, with announcements about conversational intelligence that clubs natural human language, with advanced machine intelligence. New frameworks like the Microsoft Bot Framework and Skype Bot Platform were announced and enhancements to the Cortana Intelligence Suite powered with Microsoft Cognitive Services were made which exposes Intelligence APIs that allow systems to see, hear, speak, understand and interpret our needs with natural communication.

In this article we will take a look at two of my favourites, **Microsoft Bot Framework** and **Microsoft Cognitive Services**.

Microsoft Bot Framework

As a SciFi fan, Robots and Artificial Intelligence has always fascinated me. Bots are nothing but software applications which run automated tasks. They can accept commands and perform tasks which are structured.

Microsoft Bot Framework lets you easily build Bots and connect them to various channels like Skype, Slack, Office 365 Email and many more. If you have an existing Bot, you can also connect them to various channels. This means you don't have to rewrite or build new Bots for every channel (Skype, Slack, Twitter etc) if you want to target a wider audience. You can concentrate on building better bots and let Bot Framework take care of connecting to other 3rd party services. Here is a screenshot of one of my Bot admin page.



As you can see, my bot, **Knowledge Guru**, is connected to Skype and Web Chat. This means my Bot has a face and any one can interact with it. Any Skype User can send message to this bot and I can also embed this Bot in any website. Here is the link to the Web Chat https://knowledgegurubot.azurewebsites.net/. You can ask Knowledge Guru about any Academic topic and retrieve papers published by various Authors and Affiliations within seconds. The chat control you see in this page is provided by the Bot Framework. This is very useful and saves us some time.

Note: This is just a starting point. There are bugs in this priliminary app and I will be eliminating them and making the conversation smarter as I build on it.

Bots are not new and Microsoft is not the first player in the new Bot race! Many enthusiasts including me have created automated services and have used them in our day to day life. In 2008, I created my first Bot, RemindMeAbout, a Twitter Bot which accepts commands as tweets and adds reminders to my calendar. This bot was simple and accepted only SPECIFIC commands. It was not intelligent to understand spelling mistakes or different sentences. This is where Microsoft Cognitive services comes into play.

Microsoft Cognitive services

Microsoft Cognitive services lets you add intelligence to your Apps. They are a set of services which can be used to add different intelligent capabilities like Language Understanding, Vision, Speech etc. with very little code. This means you can interact with your bot with natural language and talk to it like a human.

Vision	Speech	Language	Knowledge	Search
Computer Vision	Custom Recognition	Bing Spell Check	Academic	Bing Web Search
Emotion	Speaker Recognition	Language Understanding	Entity Linking	Bing Image Search
Face	Speech	Linguistic Analysis	Knowledge Exploration	Bing Video Search
Video		Text Analytics	Recommendations	Bing News Search
		WebLM		Bing Autosuggest

I encourage everyone to visit https://www.microsoft.com/cognitive-services/ and read through all the services. Microsoft has provided a good explanation and some helpful videos to understand each service.

If you are aware of Azure Machine Learning Services or Cortana Analytics Suit, then this service from Microsoft is not entirely new to you. It was possible to setup a Machine Learning Experiment, build a Model, run Experiments, publish it as a Service and use it in your app. But this was always a little challenging for me as I was not a Data Scientist and Machine Learning is not exactly my area of expertise. Cognitive Services makes this process easy for an everyday developer to concentrate more on building better apps with intelligent services and not to worry about setting it up.

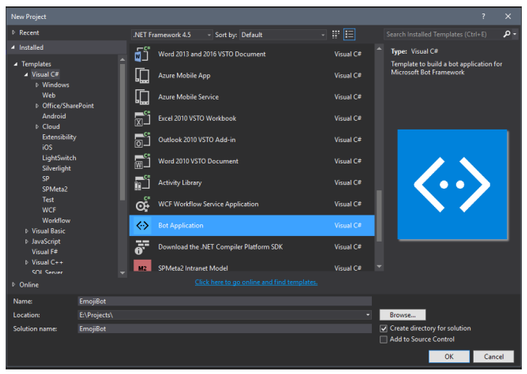
I will demonstrate this with a simple fun Bot and let's call it **Emoji Bot**. Make sure you read the documentation at https://dev.botframework.com/ and https://www.microsoft.com/cognitive-services/en-us/emotion-api - Emotion API of Microsoft

Cognitive Services. Emoji bot will accept any photo as a message and return emojis based on the emotion detected by Emotion API. Here is a screenshot

EmojiBot

Make sure your development environment is ready for Bot Framework development. We will be using the Free Visual Studio 2015 Community Edition. Visit Bot Framework Getting started page, http://docs.botframework.com/connector/getstarted/#getting-started-in-net, and download and install the necessary project template as detailed in the "Getting started in .NET" section. Make sure all Visual Studio extensions are updated before starting the project.

1. Create a new Bot Application using Visual Studio 2015



2. Replace the auto generated code for Post method in MessageController.cs with the following code

```
if (message.Type == "Message")
{
 //Get attachment
 if (message.Attachments.Count > 0)
 {
  string imageUri = message.
  Attachments[0].ContentUrl;
  return message.
  CreateReplyMessage(await Utilities.
  CheckEmotion(imageUri));
 }
 else
 {
  return message.
 CreateReplyMessage("Send me a photo!");
 }
}
else
{
 return HandleSystemMessage(message);
}
```

In the code, we check the number of attachments. Images sent in a message are retrievable through `Attachments` property. We then use this image to detect faces and their respective emotions in step 3.

3. Add the following code to MessageController.cs file:

```
public static class Utilities
{
 public static async Task<string>
 CheckEmotion(string query)
 {
var client = new HttpClient();
var queryString = HttpUtility.
ParseQueryString(string.Empty);
string responseMessage = string.Empty;

// Request headers
client.DefaultRequestHeaders.
Add("Ocp-Apim-Subscription-Key", "YOUR
SUBSCRIPTION KEY");

// Request parameters
var uri = "https://api.projectoxford.
ai/emotion/v1.0/recognize";
HttpResponseMessage response = null;
byte[] byteData = Encoding.UTF8.
GetBytes("{ 'url': '" + query + "'
}");

using (var content = new
ByteArrayContent(byteData))
{
 content.Headers.ContentType = new
 MediaTypeHeaderValue("application/
 json");
 response = await client.
 PostAsync(uri, content).
 ConfigureAwait(false);
}

string responseString = await
response.Content.ReadAsStringAsync();

EmotionResult[] faces = JsonConvert.
DeserializeObject<EmotionResult[]>
(responseString);
if (faces.Count() > 0)
{
 foreach (EmotionResult r in faces)
 {
  string emotion = new[]
  {
   Tuple.Create(r.scores.contempt,
   "Contempt"),
   Tuple.Create(r.scores.disgust,
   "Disgust"),
   Tuple.Create(r.scores.fear,"Fear"),
   Tuple.Create(r.scores.happiness,
   "Happiness"),
   Tuple.Create(r.scores.neutral,
   "Neutral"),
   Tuple.Create(r.scores.sadness,
   "Sadness"),
   Tuple.Create(r.scores.surprise,
   "Surprise"),
  }.Max().Item2;
  responseMessage += $"![{emotion}]
  (https://emojibot.azurewebsites.net/
  {emotion}.png)";
 }
  }
  else
  {
   responseMessage = "No faces detected";
  }
  return responseMessage;
 }
}
```

In this code, we pass the image to Emotion API to detect faces. If faces are found, then their emotional scores are sorted. We can then use this value to reply with an appropriate emoji. Make sure you get your subscription key by visiting https://www.microsoft.com/cognitive-services/en-US/subscriptions

4. Add the following classes to MessageController.cs

```
public class Scores
{
 public double anger { get; set; }
 public double contempt { get; set; }
 public double disgust { get; set; }
 public double fear { get; set; }
 public double happiness { get; set; }
 public double neutral { get; set; }
 public double sadness { get; set; }
 public double surprise { get; set; }
}

public class EmotionResult
{
 public Scores scores { get; set; }
}
```

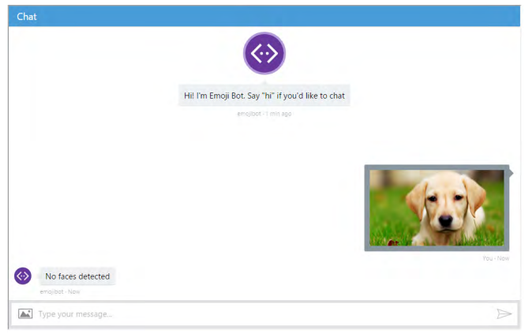
Make sure you also update your AppId and Secret in app.config after creating a bot in https://dev.botframework.com/#/bots .

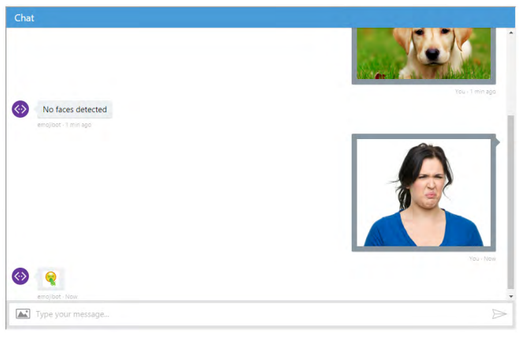
Once the above steps are completed, Enable Web channel and test your bot by adding an iframe to the default.html file.

```
<iframe src='https://webchat.
botframework.com/embed/
emojibot?s=<YOUR SECRET HERE'
style="height:600px;width:50%;"></
iframe></body>
```

That's it! Publish your bot to Azure and play around. You can also play around with Emoji Bot at https://emojibot.azurewebsites.net/

Here are some more screenshots.





As you can see, in a short time we were able to build a simple intelligent bot. You can further improve the bot by replacing faces in the image with real emojis.

Microsoft Bot Framework and Cognitive Services are still evolving and I noticed that there are some bugs and sometimes messages are not delivered or delivered late, but I am sure they will be fixed soon. Try out other APIs in Cognitive Services for more interesting ideas ■

Download the entire source code from our GitHub Repository at bit.ly/dncm24-msbot-framework

About the Author

shoban kumar

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*

SOLID PRINCIPLES
DEPENDENCY INVERSION

Finally, we are at the end of a long journey on SOLID principles. As a quick review, you can find links to the previous articles at http://www.dotnetcurry.com/tutorials/software-gardening/. In this episode, we tackle Dependency Inversion or as it is commonly called, Dependency Injection (DI).

As is always required when talking about SOLID, here's how Uncle Bob defines it in his book "Agile Principles, Patterns, and Practice in C#":

> *A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
>
> *B. Abstractions should not depend upon details. Details should not depend upon abstractions.*

In other words, don't have a lot of dependencies in your code. Dependencies make code more fragile and harder to test and maintain. The idea is to create an instance of the dependency, then inject that instance into the module.

If you use the `new` keyword to create an instance of a class, you may have a good candidate for dependency injection.

The most common way, and generally the most recommended way, to implement DI, is with "constructor injection". Let's look at a simple explanation of this using what's called "poor man's dependency injection". This code shows a common scenario of new-ing up things that you need, in this case it's a logger.

```
class Program
{
 static void Main(string[] args)
 {
  var processor = new GizmoProcessor();
 }
}

public class GizmoProcessor
{
 public void Process()
 {
  // do something
  var logger = new TextLogger();
  logger.WriteLogMessage("Something
  happened");
 }
}

public class TextLogger
{
 public void WriteLogMessage(string
 Message)
 { }
}
```

Here's the modified code using Constructor Injection.

```
class Program
{
 static void Main(string[] args)
 {
  var processor = new GizmoProcessor(new
  TextLogger());
 }
}

public class GizmoProcessor
{
 private readonly ILogger _logger;
 public GizmoProcessor(ILogger logger)
 {
  _logger = logger;
 }

 public void Process()
 {
  // do something
 _logger.WriteLogMessage("Something
happened");
    }
}

public class TextLogger : ILogger
{
 public void WriteLogMessage(string
 Message)
 { }
}

public interface ILogger
{
 void WriteLogMessage(string Message);
}
```

The difference is `TextLogger` is instantiated in the calling program, then passed in as a parameter. This makes it easy to change out the `ILogger` implementation and unit test `GizmoProcessor` because the dependency is removed. It's called poor-man's because there is no DI framework nor Inversion of Control (IoC) container to handle the injection for you automatically.

Let's look at a more real-world example of Dependency Injection. I will build slowly from an out-of-the-box ASP.NET MVC 5 solution to using DI and an IoC container to improve on the code. An IoC container is a framework that handles the dependency injection details for you. I will not fully build out this application, but only look at a small part of it.

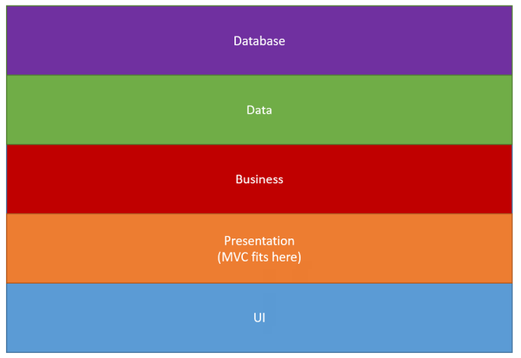
The example starts with `CustomersController`. I include the entire class here so you can compare it to changes we'll make to it later.

```
public class CustomersController :
Controller
{
 private InjectionContext db = new
 InjectionContext();

 public ActionResult Index() {
  return View(db.Customers.ToList());
 }
```

```
public ActionResult Details(int? id)
{
 if (id == null)
 {
  return new
  HttpStatusCodeResult(HttpStatusCode.
  BadRequest);
 }
Customer customer = db.Customers.
Find(id);
if (customer == null)
{
 return HttpNotFound();
}
return View(customer);
}

public ActionResult Create()
{
 return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult
Create([Bind(Include = "Id,FirstName,
LastName,Street1,Street2,City,State,
PostalCode,Country,Phone")] Customer
customer)
{
 if (ModelState.IsValid)
 {
  db.Customers.Add(customer);
  db.SaveChanges();
  return RedirectToAction("Index");
 }
 return View(customer);
}

public ActionResult Edit(int? id)
{
 if (id == null)
 {
  return new HttpStatusCodeResult
  (HttpStatusCode.BadRequest);
 }
 Customer customer =
 db.Customers.Find(id);
 if (customer == null)
 {
  return HttpNotFound();
 }
 return View(customer);
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult
Edit([Bind(Include = "Id,FirstName,
LastName,Street1,Street2,City,State,
PostalCode,Country,Phone")]
Customer customer)
{
 if (ModelState.IsValid)
 {
  db.Entry(customer).State =
  EntityState.Modified;
  db.SaveChanges();
  return RedirectToAction("Index");
 }
 return View(customer);
}

public ActionResult Delete(int? id)
{
 if (id == null)
 {
  return new HttpStatusCodeResult
  (HttpStatusCode.BadRequest);
 }
 Customer customer =
 db.Customers.Find(id);
 if (customer == null)
 {
  return HttpNotFound();
 }
 return View(customer);
}

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int
id)
{
 Customer customer = db.Customers.
 Find(id);
 db.Customers.Remove(customer);
 db.SaveChanges();
 return RedirectToAction("Index");
}

protected override void Dispose(bool
disposing)
{
 if (disposing)
 {
  db.Dispose();
 }
 base.Dispose(disposing);
}
}
```

Let's learn what we can from this piece of code. First, we see the data context gets instantiated. This binds the context to the controller in a strong way. The context is a dependency and what if it changes? The controller knows a lot about the context. In fact, too much. Further, how will you unit test this? You can't easily remove this hard dependency on the database.

The next thing is related and goes to the very definition of MVC: Model, View, Controller and how it is related to the standard three-tier application with UI, Business, and Data layers. The View is not the UI. It is the information that is displayed. The Model is not the database. It is the representation of the data needed by the View. The Controller is the thing that hooks up the View and the Model and should be a very thin layer, meaning it does very little. Taking this an additional step, the Data layer is not the database. It handles the data access, creating SQL statements, etc. In a standard MVC application, this is where Entity Framework sits. So, instead of three layers, we have many.



What this tells us is that we need to get data access and business rules out of the controller. To do this, we'll create a `CustomerService` class that sits in the Business layer.

```
public class CustomerService :
ICustomerService
{
 private InjectionContext db = new
 InjectionContext();

 public IList<Customer>
 GetModelForIndex()
 {
  return db.Customers.ToList();
 }

 public Customer GetModelForDetails(int?
 id)
 {
  return db.Customers.Find(id);
 }

 public Customer GetModelForEdit(int?
 id)
 {
  return db.Customers.Find(id);
 }

 public Customer GetModelForDelete(int?
 id)
 {
  return db.Customers.Find(id);
 }

 public void Delete(int id)
 {
  Customer customer = db.Customers.
  Find(id);
  db.Customers.Remove(customer);
  db.SaveChanges();
 }

 public bool CanSaveCustomer(Customer
 model)
 {
  if (IsValid(model))
  {
   db.Customers.Add(model);
   db.SaveChanges();
   return true;
  }
  return false;
 }

 private bool IsValid(Customer model)
 {
  return true;
 }
}
```

Here's the complete Controller we now have.

```
public class CustomersController :
Controller
{
 private ICustomerService service = new
 CustomerService();

 public ActionResult Index()
 {
  return View(service.
  GetModelForIndex());
 }

 public ActionResult Details(int? id)
 {
  if (id == null)
  {
```

```
  return new HttpStatusCodeResult
  (HttpStatusCode.BadRequest);
 }

 var model = service.
 GetModelForDetails(id);

 if (model == null)
 {
  return HttpNotFound();
 }
 return View(model);
}

public ActionResult Create()
{
 return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult
Create([Bind(Include =
      "Id,FirstName,LastName,Street1,
Street2,City,State,PostalCode,
Country,Phone")] Customer model)
{
 if (service.CanSaveCustomer(model))
 {
  return RedirectToAction("Index");
 }
 return View(model);
}

public ActionResult Edit(int? id)
{
 if (id == null)
 {
  return new HttpStatusCodeResult
  (HttpStatusCode.BadRequest);
 }
 var model = service.
                GetModelForEdit(id);
 if (model == null)
 {
  return HttpNotFound();
 }
 return View(model);
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include
= "Id,FirstName,LastName,Street1,
Street2,City,State,PostalCode,Country,
Phone")] Customer model)
{
 if (service.CanSaveCustomer(model))
 {
  return RedirectToAction("Index");
 }
 return View(model);
}

public ActionResult Delete(int? id)
{
 if (id == null)
 {
  return new HttpStatusCodeResult
  (HttpStatusCode.BadRequest);
 }

 var model = service.GetModelForDelete
 (id);

 if (model == null)
 {
  return HttpNotFound();
 }

 return View(model);
}

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed
(int id)
{
 service.Delete(id);
  return RedirectToAction("Index");
 }
}
```

If you're now saying, "That's great, but you've swapped out the dependency on `InjectionContext` with a dependency on `CustomerService`", you're correct. As with any refactoring, do this in small steps. The next step is to hook up dependency injection so that whenever this controller gets instantiated, an instance of the `CustomerService` class will be injected into the controller. The key to this is the use of the `ICustomerService` interface.

The first thing to add is a constructor to the controller and a field to hold the `CustomerService` instance.

```
private readonly ICustomerService service;

public CustomersController
(ICustomerService svc)
{
 service = svc;
}
```

Now, add a Dependency Injection library. I'm going to use Ninject and the Ninject.MVC5 extension. Install the Ninject.MVC5 package from NuGet. It will install everything you need. You can learn about Ninject at http://www.ninject.org/ and the MVC5 extension at https://github.com/ninject/ninject.web.mvc/wiki.

After installing the Ninject assemblies, you need to tell Ninject how to resolve the dependency. In the App_Start folder for the solution, you'll find a new file named NinjectWebCommon.cs. Open it and navigate to the RegisterServices method. Add the `CustomerService`.

```
private static void
RegisterServices(IKernel kernel)
{
 kernel.Bind<ICustomerService>().
 To<CustomerService>();
}
```

You can now run your application and Ninject will automatically inject an instance of the `CustomerService` class. When you unit test, you'll create a mock instance of `ICustomerService` and pass it in. It will look something like this:

```
ICustomerService mockCustSvc = new
MockCustomerService();
var CustomersController = new
CustomersController(mockCustSvc);
```

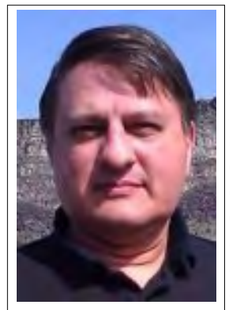
The next step you should take is to remove the dependency on `InjectionContext` that exists in `CustomerService`. The best way to do this is using the RepositoryPattern then inject an IRepository implementation. I leave that for you to figure out.

So, I've shown you how to get started with dependency injection. A word of warning: it can get very, very complicated and can make it harder to debug because you don't always know what's happening due to the class that gets injected, can change. I know several top-notch, well-known .NET developers who don't like IoC containers. Like any solution to a problem, your mileage may vary.

However, using dependency injection does solve the issue of tightly coupling. It also makes your code more testable and easier to change out implementations. Now that you've got a good basic understanding of SOLID, I highly recommend the book "Adaptive Code via C#" by Gary McLean Hall and published by Microsoft Press.

And as always, by using Dependency Injection and SOLID, you'll help ensure that your software is green, lush, and growing ■

About the Author



craig berntson

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.

TYPESCRIPT OVERVIEW

These days JavaScript is a must-have skill for any web developer. The language that was used to perform simple operations in the browsers is now used to build full-fledged web applications, APIs and even mobile applications. Using the language to build applications of such massive scale is not an easy task. The additions to the language in ECMAScript 6 (a.k.a ES2015 or ES6) and the new features of ECMAScript 7 (a.k.a ES2016), which are under development have made the language a better fit for big applications. Though these additions are great, they couldn't bring all advantages of a typical typed language to JavaScript. JavaScript was never meant to be a typed language and the TC39 committee doesn't want to take the language in that direction. In addition to this, browsers need some time to implement these features.

Microsoft's solution to these challenges is TypeScript – a superset of JavaScript. TypeScript is a open source programming language created and maintained by Microsoft and was first announced in October 2012. Let's understand what the language is about and how it works. We will be using TypeScript v 1.8.0.

What is TypeScript?

TypeScript is an open source programming language written on top of JavaScript to support types. TypeScript is not a completely new language and is not intended to replace JavaScript in anyway. It adheres to all principles of JavaScript and just adds types on top of it. If you are already good at JavaScript, it won't take you much time to learn TypeScript.

TypeScript can be used to write everything that can be written in JavaScript. With support of types, it looks very close to any other typed OOPs language like C# and Java. The types are optional, so it is not mandatory to strongly type everything. However good usage of types has many advantages. For example it enhances the productivity of the team. Strict type checking system makes the code more predictable. The type system of the language helps in solving many issues during development which could not be caught until runtime. This feature reduces the development time of a team. As most of the issues are resolved upfront, it reduces the cost of fixing the bugs later.

The TypeScript team is working very hard to keep the language updated with the latest specifications of JavaScript. Most of the proposed features of ES6 and ES7 are implemented by TypeScript. They get converted into their ES5 equivalent when the TypeScript files are *transpiled*. **This makes it possible to use the latest features of JavaScript even when the browsers have not implemented them natively**.

An existing JavaScript library need not be rewritten to be used in TypeScript. We need a type definition file declaring types for the APIs exposed by the library. The GitHub project Definitely Typed does a great job of creating and maintaining type definition files for most of the JavaScript libraries. These files are made available through a package manager to make it easier to install and use the type definition files.

Basic constructs: Types, Functions, Classes

Data Types and Variables

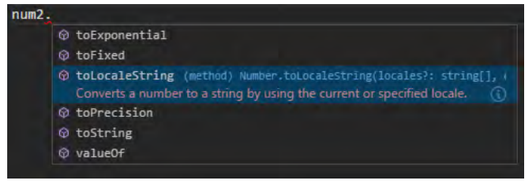
TypeScript defines a set of basic and general purpose types. They are:

- **number:** All numeric values are treated as numbers. A variable of number type gets access to all APIs of the type *Number* defined by the browser
- **string:** Any valid JavaScript string value gets string type. The string APIs defined by the browser are accessible to this type
- **boolean:** A boolean type variable can be assigned with true or false. Default value of a Boolean variable is false
- **any:** A generic type. A variable whose type cannot be represented using any of the basic types, types defined by the browser or the custom classes written in the application, can be declared using the *any* type
- **Empty object type ({}):** This type represents objects that do not have any instance members or members that cannot be added to the object. Hence these are just empty objects. This type is not used in most of the cases.

Syntax of declaring a variable is similar to the way we do it in JavaScript, except we declare type along with it.

```
var num: number;
var num2 = 10;
```

The variable *num2* infers its type based on the value assigned to it, which is number. When we try to access members of the variable *num2*, auto completion suggests the members defined in the `Number` type. Following is a screenshot from Visual Studio Code (a free code editor from Microsoft) showing auto completion when we access members of *num2*:



Declaring a variable without using a type and without assigning any value to it makes type of the variable *any*. If types of the variable and the value assigned to it differ, the TypeScript compiler generates an error.

Functions

TypeScript functions can have typed arguments

and a return type. TypeScript doesn't allow to pass variables of different types into the function unless there are explicit declarations of the function with those types.

Consider the following function and the statements that calls the function:

```
function add(val1: number, val2:
number): number{
  return val1 + val2;
}
var numResult = add(10, 20);
var strResult = add("The number is: ",
"10");
```

Out of the two statements, the second won't compile as the function expects the parameters to be numbers and we are passing strings here. Return type of the function is a number, so the variable *numResult* also gets the type number.

It is possible to make the function accept either a number or a string using union types. We need to define a union type to represent number or string and use it as the type in the function definition. Following snippet shows how to rewrite the above function using union types:

```
type myType = number | string;
function add(val1: myType, val2:
myType): myType{
  if((typeof val1 === 'number' && typeof
  val2 === 'number'))
    return val1 + val2;
  else if(typeof val1 === 'string' &&
  typeof val2 === 'string')
    return val1 + val2;
  return null;
}
var result1 = add(10, 20);
var result2 = add("The number is: ",
"10");
var result3 = add("The number is:", 10);
```

The function adds the parameters when both are either numbers or strings. If both types of both parameters are different, it returns a null. Out of the three calls made to the function, we get result in the first two instances and the third one returns null.

Classes

Classes are used to define blueprint of objects. They are used extensively in OOPs based languages. Though JavaScript didn't have direct support for classes till ES6, we were still able to create the recipes similar to classes using the *prototype* property of the objects. TypeScript had support for classes since its inception.

Classes in TypeScript can have properties and members as instance variables. Their access to the outside world can be controlled using access specifiers. The class can have a constructor to initialize its members. Arguments of the constructor can be automatically converted to instance members if an access specifier is specified in the argument. Following is an example of a class:

```
class Employee{
  private bonus: number;
  constructor(private empNo: string,
  private name: string, private salary:
  number){
    this.bonus = this.salary * 0.1;
  }

  getDetails(){
    return `Employee number is ${this.
    empNo} and name is ${this.name}`;
  }

  get Name() {
   return this.name;
  }

  set Name(name: string){
   this.name = name;
  }
}
```

As the arguments passed to the constructor are marked as private, they are made members of the class and they receive the values that are passed to the constructor. The function *getDetails* returns a string containing details of the employee.

The default access specifier in TypeScript is *public*. Here, the method *getDetails* becomes public. It can be called using any object of the class. The field `bonus` shouldn't be accessed directly using the object, so it is explicitly made private.

The property *Name* with public getter and setter blocks are encapsulating the private field *name* in the class. This is to prevent direct access to the instance member from outside of the class. And the field *bonus* just has a getter block around it which allows it to be accessed from outside the class and prevent assigning any value to it.

We can instantiate a class using the `new` keyword. Public members of the class can be accessed using the object created.

```
var emp = new Employee("E001", "Alex",
10000);
emp.Name = "Alisa";
console.log(emp.getDetails());
console.log(emp.Bonus);
```

Inheritance

Reusability is a word that we cannot ignore as programmers. We come across this word every now and then. One of the several ways in which code reusability can be achieved while dealing with classes and objects is through **inheritance**. A class can be inherited from an existing class. The new class gets members of the parent class along with its own.

The child class gets access to all *protected* and *public* members of the parent class and not to the *private* members. When a child class is instantiated, the parent class also gets instantiated. The child class can call constructor of the parent class using the *super* keyword.

Let's modify the *Employee* class defined above so that some of its members would be accessible through a child class and define a new class, *Manager* to see how inheritance works.

```
class Employee{
  private bonus: number;
  constructor(protected empNo: string,
  protected name: string, protected
  salary: number){
    this.bonus = this.salary * 0.1;
  }
  getDetails(){
    return `Employee number is ${this.
    empNo} and name is ${this.name}`;
  }
  get Name() {
   return this.name;
  }
  set Name(name: string){
   this.name = name;
  }
  get Bonus(){
    return this.bonus;
  }
}

class Manager extends Employee {
  constructor(empNo: string, name:
  string, salary: number, private
  noOfReportees: number) {
    super(empNo, name, salary);
  }
  getDetails(){
    var details = super.getDetails();
    return `${details} and has ${this.
    noOfReportees} reportees.`;
  }
}
```

In constructor of the *Manager* class, we call the constructor of the *Employee* class in the first statement. The *super* keyword in the child class refers to the current instance of the parent class. Call to the parent class constructor has to be the first statement in the child class. If the parent class has a parameterized constructor, it is mandatory to call the parent class constructor.

Also, observe the way the *getDetails* method calls the parent class method. This method overrides the parent class method. When *getDetails* is invoked using an object of the *Manager* class, it calls method of the *Manager* class. This completely hides the parent class method. But the parent class method is available for the child class to invoke, and it can be done using the *super* keyword.

Interfaces

Interfaces are used to create contracts. They don't provide a concrete meaning to anything, they just declare the methods and fields. Because of this, an interface cannot be used as-is to build anything. An interface is meant to be inherited by a class and the class implementing the interface has to define all members of the interface.

Consider the following interface:

```
interface IShape{
  area(): number;
}
```

The above interface represents a shape and declares a method to calculate surface area of the shape. It doesn't cover any details about the shape - for e.g. the kind of shape, if it has length and breadth or any other such details. These details have to be provided by the implementing class. Let's define two classes, *Square* and *Rectangle* by implementing this interface.

```
class Square implements IShape{
  constructor(private length: number){}
  get Length(){
    return this.length;
  }

  area(){
    return this.length * this.length;
  }
}

class Rectangle implements IShape{
  constructor(private length: number,
  private breadth: number){}

  area(): number{
    return this.length * this.breadth;
  }
}
```

We can instantiate these classes and assign them to references of the interface type. We can access the members declared in the interface using this instance.

```
var square: IShape = new Square(10);
var rectangle: IShape = new
Rectangle(10, 20);
console.log(square.area());
console.log(rectangle.area());
```

The class *Square* class has an additional property *Length* that is not declared in the interface. Though the object *square* is an instance of *Square* class, we cannot access the members that are defined in the class and not in the interface. However, the object *square* can be casted to the type *Square* and then we can use the members defined in the *Square* class

```
var squareObj = square as Square;
console.log(squareObj.Length);
```

Decorators

Decorators are used to extend the behavior without modifying the implementation. Decorators can be applied on classes, members of classes, functions or even on arguments of function. It is a feature proposed for ES7 and is already in use by some of the JavaScript frameworks including Angular 2.

Creating and using decorators is very easy. A custom decorator is a function that accepts some arguments containing details of the target on which it is applied. It can modify the way the target works using this information.

The following snippet defines and uses a decorator:

```
function nonEnumerable(target, name,
descriptor){
  descriptor.enumerable = false;
  return descriptor;
}
class Person {
  fullName: string;
  @nonEnumerable
  get name() { return this.fullName; }
  set name(val) {
    this.fullName = val;
  }
  get age(){
    return 20;
  }
}
var p = new Person();
for(let prop in p){
  console.log(prop);
}
```

The decorator *nonEnumerable* sets enumerable property of a field to `false`. After this, the property won't be encountered when we run a *for...in* loop on the object. The loop written at the end of the snippet prints only the property 'age '.

Modules

Writing applications consisting of hundreds of files is almost impossible without a modular approach. As TypeScript was created to solve the issues with creating large applications using JavaScript, support for modules was added to it. After announcement of the ES6 module system, the module system of the language was renamed to namespaces and ES6 module system is now a first class citizen in TypeScript.

ES6 module system treats every file as a module. A module can export the objects that it wants to make them available to the other modules and also import the objects exported by other modules. In a typical application using ES6 modules, one module made responsible to perform initial task of the application, loads a few other modules of the application. These modules in turn load other modules in the application and so on.

A module can export any number of functions, classes or variables. By default, the objects are exported with their original names. We can change this, if required. A module can have a default exported member as well. Following snippet shows examples of different export statements:

```
//Inline export with same name
export class MyClass{}
export function myFunc(){}

//Exporting a group of members
export {
  MyClass,
  myFunc
};

//Rename while exporting
export {
  MyClass as AnotherClass,
  myFunc as anotherFunc
};

//Default export
export default myMemberToExport;
```

When a module imports another module, it may import all exported members, some of them or none at all. The importing module also has the option to rename the importing object. The following snippet shows examples of different import statements:

```
//Importing all exported objects
import * as module1 from "./module1";
//Importing selected objects
import {MyClass1, MyClass2} from "./
module1";

//Importing selected objects and rename
them
import {MyClass1 as Module1MyClass1,
MyClass2 as Module1MyClass2} from "./
module1";


//Importing default export object
import d from "./module1";
```

Transpiling and using on a web page

TypeScript is not created for the browsers. It is a language to be used by developers to write code. Hence we need to transpile the TypeScript code to JavaScript to be able to run it on the browser. There are several ways in which TypeScript code can be transpiled to JavaScript, each of them can be used in different scenarios depending upon the need.

Transpiling on the Fly

The easiest way to try TypeScript is by transpiling it on the fly. This approach works for demos really well, but it is not the recommended way for production. Let's see how on-the-fly transpiler works before jumping into pre-compiled script loading. To run TypeScript directly in the browser, we need the following libraries:

- **TypeScript transpiler:** a JavaScript file that takes care of transpiling TypeScript on the browser

- **SystemJS module loader:** SystemJS is a universal module loader. It loads all types of JavaScript modules. It is used to load ES6 modules without transpiling them

- **SystemJS polyfill:** used to polyfill the missing module systems. For example, if SystemJS is used to load an AMD style module and no AMD library is loaded on the page, then it polyfills the AMD module system

- **ES6 Shim:** It is used to polyfill ES6 promises in unsupported browsers

The HTML page running TypeScript code has to refer to the four libraries we just discussed. Following is the content of an HTML page designed for this purpose:

```
<!DOCTYPE html>
<html>
<head>
 <title>Transpiling TypeScript in the
 browser</title>

 <meta name="viewport"
 content="width=device-width, initial-
 scale=1">

 <script src="https://cdnjs.cloudflare.
 com/ajax/libs/es6-shim/0.35.0/es6-shim.
 min.js"></script>

 <script src="https://cdnjs.cloudflare.
 com/ajax/libs/systemjs/0.19.25/system-
 polyfills.js"></script>

 <script src="https://cdnjs.cloudflare.
 com/ajax/libs/systemjs/0.19.25/system.
 js"></script>

 <script src="https://npmcdn.com/
 typescript@1.8.9/lib/typescript.js">
 </script>

</head>
<body>

<div>Area of the square is: <span
id="areaOfSquare"></span></div>

<div>Area of the rectangle is: <span
id="areaOfRectangle"></span></div>

</body>
</html>
```

The TypeScript file has the classes *Square* and *Rectangle* we saw earlier in an example, and both of them calculate area. The file exports these two classes. Following is the content of this file:

```
export class Square{
  constructor(private length: number){}
  area(){
    return this.length*this.length;

  }
}

export class Rectangle{
  constructor(private length: number,
private breadth: number){}

  area(): number{
    return this.length * this.breadth;
  }
}
```

It is saved in a file named app.ts. We need to load this file using System.js in the index.html and use the exported members. Following snippet shows how to load the file and use the exported members:

```
System.config({
  transpiler: 'typescript',
  typescriptOptions: {
  emitDecoratorMetadata: true },
  packages: {'.': {defaultExtension:
  'ts'}}
});

System.import('app')
.then(function(result) {
  var s = new result.Square(10);
  var r = new result.Rectangle(10, 20);
  document.querySelect("#areaOfSquare").
  innerHTML = s.area();
  document.querySelector
  ("#areaOfRectangle").innerHTML =
  r.area();
});
```

This page can't be loaded in the browser directly. It has to be loaded from a server. To start a server easily, you can use the global npm *http-server* to start a Node.js server and open the URL depending upon the port number where it starts the server. Setting up Node and starting a server is covered in this article http://www.dotnetcurry.com/nodejs/1203/create-web-server-nodejs.

Pre-transpiling using command line

TypeScript can be pre-compiled to JavaScript using a number of approaches like using command line interface (CLI) of TypeScript, using task runners such as Grunt or Gulp, or using module loaders like Webpack and JSPM. Let us see how to pre-compile using the command line interface.

To use TypeScript's CLI, we need to install the global npm package for TypeScript using the following command:

```
> npm install -g typescript
```

After successful completion of this command, we can use the `tsc` command to transpile the TypeScript files.

```
> tsc app.ts
```

As we are using ES6 style modules, we need to tell TypeScript the target module format we are looking for. TypeScript converts the modules into CommonJS format if no module system is specified. If we need to convert into any module type other than CommonJS, we need to specify it in the command line option. Following command converts the module to SystemJS:

```
> tsc app.ts --module system
```

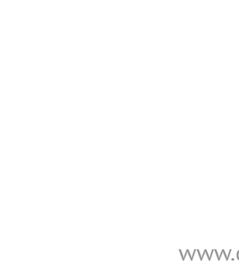
This command produces a file named app.js. We can load this file in the HTML file using the SystemJS module loader. We need to make some minor changes to the SystemJS code we wrote earlier to load this file. Following is the modified code:

```
System.config({
  packages: {'.': {defaultExtension:
'js'}}
});

System.import('app')
.then(function(result){
  var s = new result.Square(10);
  var r = new result.Rectangle(10, 20);
  document.
    querySelector("#areaOfSquare").
    innerHTML = s.area();
  document.
    querySelector("#areaOfRectangle").
    innerHTML = r.area();
});
```

Conclusion

TypeScript makes the experience of working with JavaScript better. The language is thoughtfully designed to not invent anything new and yet make the experience of programmers working on JavaScript better ■

About the Author



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.

Image: azure.microsoft.com

Migrating from

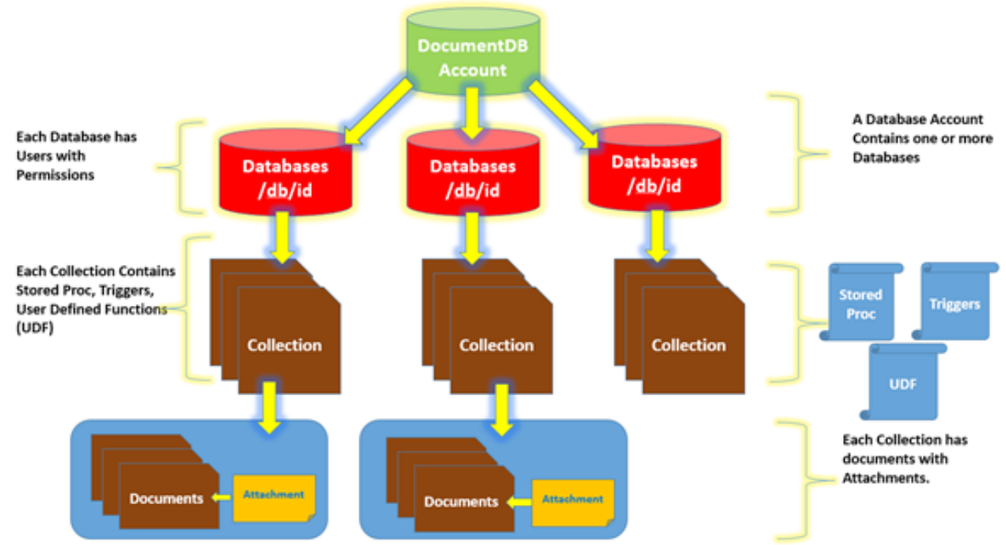
SQL Server to NoSQL

using Azure DocumentDB

RDBMS have held the bulk of the world's enterprise data for over three decades now. For developers working on the Microsoft stack, SQL Server is clearly the de facto relational database server of choice. However over the past few years, RDBMS have been struggling to keep up with the variety and volume of data exchanged in this modern web application tsunami. The type and format of the data uploaded from such web applications varies and is mostly schema independent. In such cases, the data store must also be schema-free and more importantly, flexible. That's where developers have shown interest in schema-free No-SQL Databases.

DocumentDB, by Microsoft, is a schema-free NoSQL Database. It is a fairly simple, fast database with the capability of data read/write operations with SQL Query support. DocumentDB is designed for modern web and mobile applications. If you are new to DocumentDB, please read my previous article - **Working with NoSQL using Azure DocumentDB** at http://www.dotnetcurry.com/windows-azure/1262/documentdb-nosql-json-introduction .

To use DocumentDB, we must have an active Microsoft Azure Subscription. The subscription can be purchased or a free trial subscription can be used. To use the DocumentDB Service, we must create a DocumentDB Account. The resource model of DocumentDB can be represented using the following image:



A DocumentDB Account can have multiple Databases in it. Each database has users and permissions. The Database further contains *collections*. Each collection can have Stored Procedures, Triggers and User Defined Functions (UDF). The collection contains actual Document data. This is stored in JSON form.

The data document in DocumentDB can be created using following mechanisms:

- Microsoft Azure Portal
- Using Data Migration Tool
- Client SDK for .NET, JavaScript and Node.js

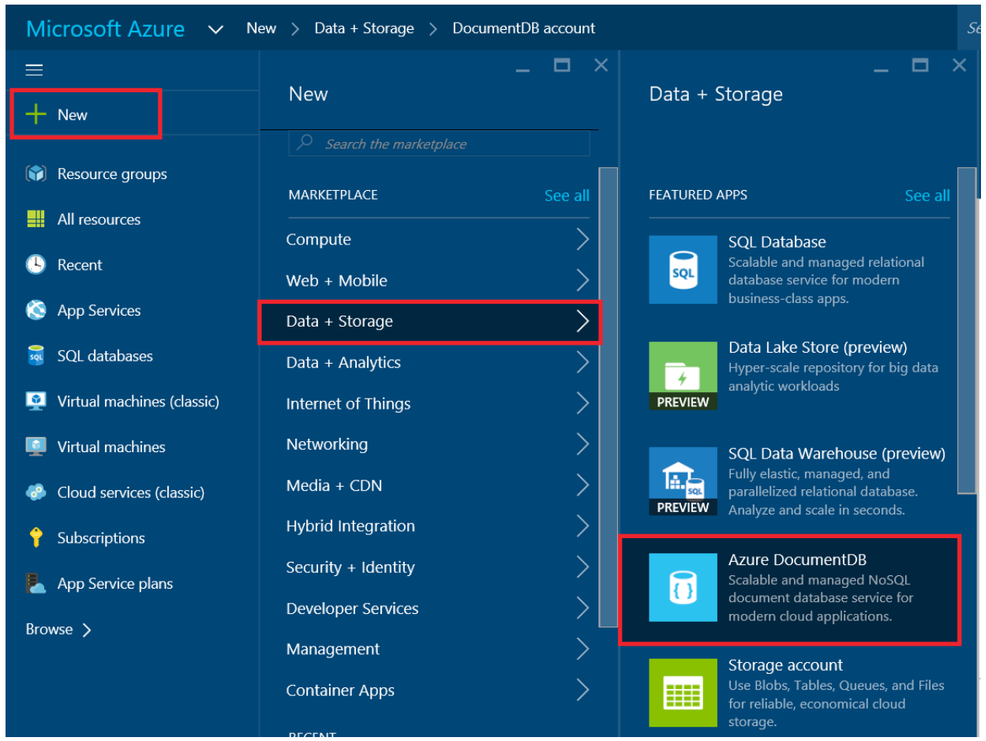
In this article, we will use Microsoft Azure Portal to create a DocumentDB Account and Database. To create collection in this database, we will make use of the **DocumentDB Data Migration Tool**.

Using the DocumentDB Data Migration Tool

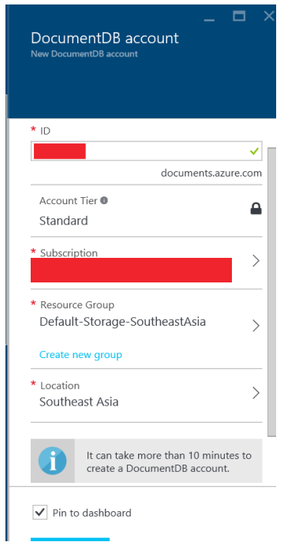
The DocumentDB Data Migration Tool is an **open source project** and can be downloaded from this link. This tool is used to migrate data from various data sources into DocumentDB. Some of the data sources supported by this tool includes SQL Server, CSV files, JSON files, MongoDB, Azure Table Storage, HBase, Amazon DynamoDB, and DocumentDB.

For our example, we will use SQL Server Data Source to migrate data to DocumentDB. When we use the SQL Server Data source, we need to make some changes in the relational data form, so as to easily migrate it to DocumentDB. Typically, these changes includes creation of an **id** field property for the primary key, reading data from the source table/view based on condition (where clause), and so on. We will implement these changes in our next steps.

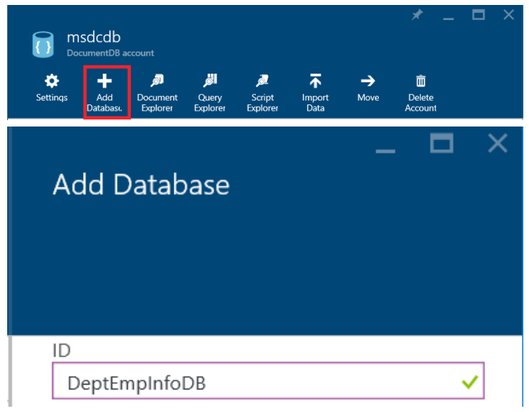
Step 1: Visit Microsoft Azure Portal, login with your credentials. Create a DocumentDB account using **New > Data+Storage > Azure DocumentDB** panel as shown in the following image:



Enter DocumentDB Account Details as shown in the following image:



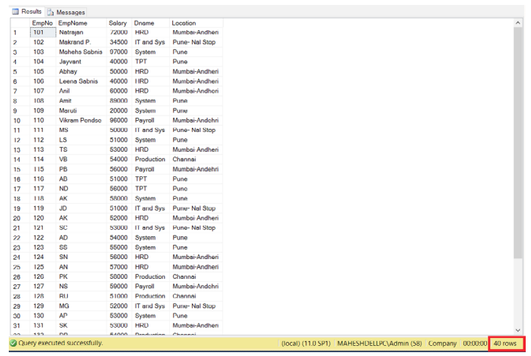
Add the database in the account using the **Add Database** option as shown in the following image:



Step 2: Once the account is created, it's time to make some changes in SQL Server as discussed earlier. We will make use of a database created using the following script:

```
Create Database Company
USE [Company]
GO

/****** Object:  Table [dbo].
[Department]    Script Date: 1/6/2016
11:23:03 AM ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

SET ANSI_PADDING ON
GO

CREATE TABLE [dbo].[Department](
  [DeptNo] [int] NOT NULL,
  [Dname] [varchar](50) NOT NULL,
  [Location] [varchar](50) NOT NULL,
 CONSTRAINT [PK_Department] PRIMARY KEY
 CLUSTERED
 (
   [DeptNo] ASC
 )WITH (PAD_INDEX = OFF, STATISTICS_
NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS =
ON) ON [PRIMARY]
) ON [PRIMARY]

GO

SET ANSI_PADDING OFF
GO

USE [Company]
GO

/****** Object:  Table [dbo].[Employee]
Script Date: 1/6/2016 11:23:13 AM
******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

SET ANSI_PADDING ON
GO

CREATE TABLE [dbo].[Employee](
  [EmpNo] [int] NOT NULL,
  [EmpName] [varchar](50) NOT NULL,
  [Salary] [int] NOT NULL,
  [DeptNo] [int] NOT NULL,
 CONSTRAINT [PK_Employee] PRIMARY KEY
      CLUSTERED
 (
    [EmpNo] ASC
 )WITH (PAD_INDEX = OFF, STATISTICS_
NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS =
ON) ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_PADDING OFF
GO

ALTER TABLE [dbo].[Employee]  WITH CHECK
ADD  CONSTRAINT [FK_Employee_Department]
FOREIGN KEY([DeptNo])
REFERENCES [dbo].[Department] ([DeptNo])
GO

ALTER TABLE [dbo].[Employee] CHECK
CONSTRAINT [FK_Employee_Department]
GO

USE [Company]
GO

/****** Object:  View [dbo].[DeptEmp]
Script Date: 1/6/2016 11:23:35 AM
******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE VIEW [dbo].[DeptEmp]
AS
SELECT dbo.Employee.EmpNo, dbo.Employee.
EmpName, dbo.Employee.Salary, dbo.
Department.Dname, dbo.Department.
Location
FROM dbo.Department INNER JOIN
  dbo.Employee ON dbo.Department.DeptNo
  = dbo.Employee.DeptNo

GO
```

This SQL Server Company database contains Department and Employee tables and a DeptEmp View. This view shows data based on a JOIN of two tables. Insert some test data in Department and Employee tables. Once the Select Query in this View is processed, the data can be viewed as shown here.



As you can see, we have 40 rows. Here each row from the above result can be mapped with a JSON document in Document DB. We need to export data from SQL server, as well as shape our hierarchical JSON document. In our case, we will use the Department Name (Dname) as document and other columns like EmpName, Salary and Location as hierarchical properties of each document. To implement this we will define the shape using the following script.

```
select cast(EmpNo as varchar) as id,
  EmpName as employeeName,
  Salary as salary,
  Location as workLocation
from DeptEmp
where Dname = 'HRD'
```

Since the JSON document has the id property as string, we are casting the EmpNo to id as string. The result returned will be as shown in the following image:

Results | Messages

	id	employeeNa...	salary	workLocation
1	101	Natrajan	72000	Mumbai-Andheri
2	105	Abhay	50000	Mumbai-Andheri
3	106	Leena Sabnis	46000	Mumbai-Andheri
4	107	Anil	60000	Mumbai-Andheri
5	113	TS	53000	Mumbai-Andheri
6	120	AK	52000	Mumbai-Andheri
7	124	SN	56000	Mumbai-Andheri
8	125	AN	57000	Mumbai-Andheri
9	131	SK	53000	Mumbai-Andheri

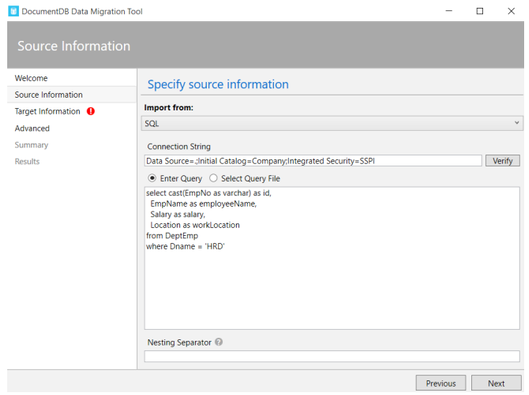
Number of rows returned are 9.

Note: *Feel free to use the Adventure Works Database* http://msftdbprodsamples.codeplex.com/releases/view/37304

Step 3: Unzip the Data Migration Tool to reveal two executable files

- dt.ext - command line utility.
- dtui.exe - UI which will provide the wizard for migration.

Step 4: Start the tool, click 'Next' on the Welcome page. On this page, we need to select the Source Information. The **Import from** dropdown shows all available option. Select SQL from the dropdown list. You should now see the **Connection String** textbox. Enter the connection string. Click on the **verify** button to verify the connection string. Select **Enter Query** radio button and enter the query in it as shown in the following image



Click on Next.

This is the **Target Information** window where we need to set the information of the DocumentDB target. Here **Export to** drop down shows options to export to DocumentDB or JSON file. This option has the following parts:

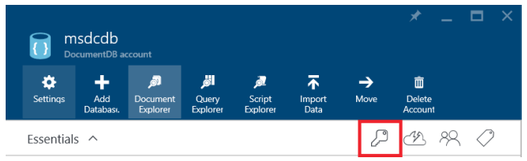
DocumentDB Bulk import - Here the migration tool creates a stored procedure, which is called by the migration tool to import data in batches. Once the import is completed, the stored procedure is released.

DocumentDB Sequential Record import - The data from the data source is imported record-by-record.

JSON File - This export contains array of JSON documents in JSON file. The data from the source is exported to this file to store data in JSON form.

Select **DocumentDB Bulk Import**. Now we need to enter the DocumentDB Connection string. This string has a unique format and we need to copy it from the Azure portal. This is a connection point to the DocumentDB Account, so we need to manually add the database name to the connection string.

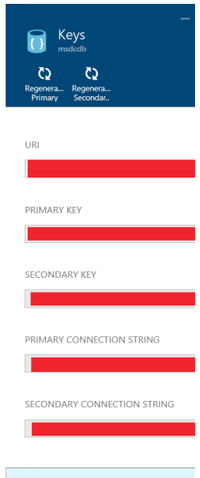
In the portal for the DocumentDB account panel, click on 'key' icon as shown in the following image:



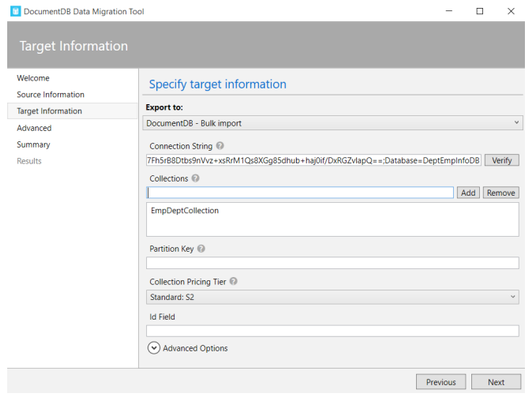
This will open the **Keys** panel, where we have the following:

- URI - Endpoint URL of the account to be connected
- PRIMARY KEY - authorization key used by the client application to connect to DocumentDB using REST and .NET, JavaScript, Node.js client SDKs. This is like UserName and Password for the account.
- SECONDARY KEY
- PRIMARY CONNECTION STRING - includes the Endpoint URL and authentication information. We need the primary connection string in our case.
- SECONDARY CONNECTION STRING

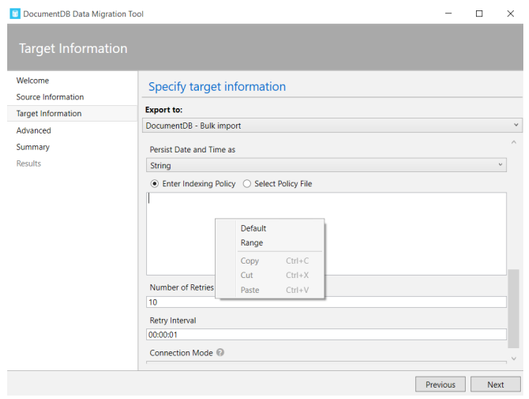
The following image shows the Keys panel:



Copy the **PRIMARY CONNECTION STRING** and paste it in the **Connection String** textbox of the migration tool. Here if we click the **Verify** button, we will get an error because the database name is not specified. Add the Database name at the end of the connection string after (;) e.g. Database=DeptEmpInfoDB. If the verify button is clicked, a successful connection message will be displayed. In the **Collection** textbox, enter the collection name. Here we can enter a name as per our choice. If it is not already present in the DocumentDB database, it will be created. Add a name and click on the **Add** button. The **Partition Key** textbox is needed if we want to put the exported data into multiple collections. The **Collection Pricing Tier** can be chosen as per our need. We have S1, S2, S3, and S1 is the cheapest. The **Id Field** textbox can be set with the property from the source to be used as **id** property, but in our case, we have already set the id property in the query. The following image shows the data entered so far

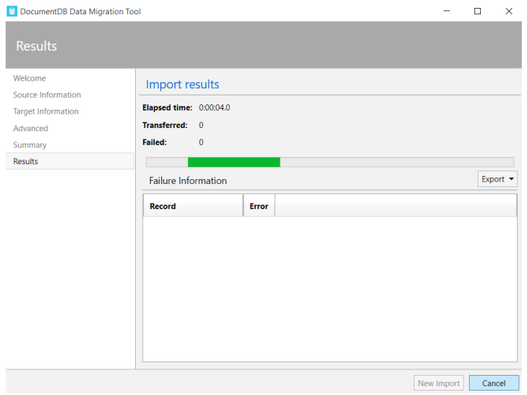


Click on the **Advanced Options**. From here we can select the Indexing policy. This policy is based on the Where, Order By clause in the query. Right-click on the **Enter Indexing Policy** textbox and select the **Range** option as shown in the following image

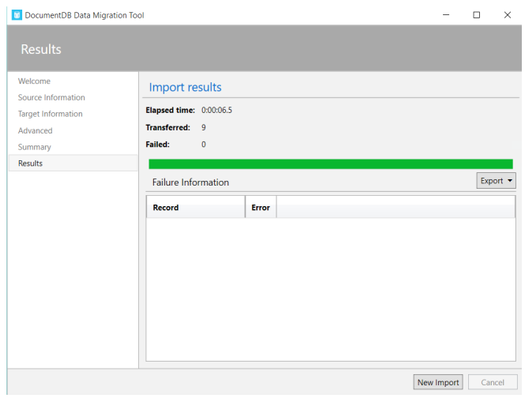


Click on 'Next'. In the Advanced page, click again on 'Next' to see a summary. In the Summary page, click on the **Import** button. This will start the data

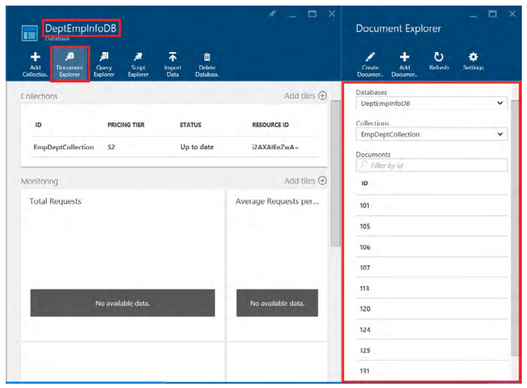
importing process as shown in the following image:



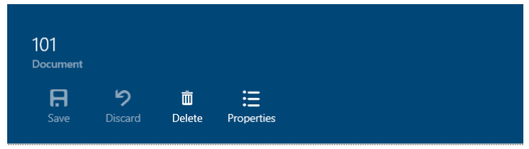
Once the import is complete, the following result will be displayed:



Step 5: Go back to the portal and in the **DeptEmpInfoDB** database, you will find that the **EmpDeptCollection** collection is created with the exported data in the document explorer.



We can see that 9 records were imported from SQL Server database. Click on any Document record (e.g. 101) and a JSON document will be displayed as shown in the following image.




```
{
  "id": "101",
  "employeeName": "Natrajan",
  "salary": 72000,
  "workLocation": "Mumbai-Andheri"
}
```

And that's how it is done. FYI, the upcoming SQL Server 2016 has built-in JSON support.

Conclusion:

Moving from a Relational database to NoSQL is a difficult decision to make and involves many decision points. But once you have decided, tools like the Azure DocumentDB Data Migration tool, comes in handy to export-import data from any data source to DocumentDB.

Download the entire source code from our GitHub Repository at bit.ly/dncm24-sql-nosql

About the Author

mahesh sabnis

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

Microsoft® Most Valuable Professional

TEST YOUR
C# BASICS

Let's have some fun while testing your C# basics around variables, types and operators. This quiz is aimed at developers who have some experience with C#. Note down the correct answers and post it over here (www.dotnetcurry.net/s/dnc-art-csharp-basics). **Some questions have more than one correct answers.** To make it more exciting, **do not** use a compiler or tool like Visual Studio to test these examples. Instead use your existing C# knowledge to choose the best option. Make sure you also post the reason why you feel a particular answer is correct.

We will be posting the correct answers and its explanation on May 20th at www.dotnetcurry.net/s/dnc-art-csharp-basics

1. Which of the following variable names are allowed? (Choose all that apply)

a. `2name`
b. `_name2`
c. `@string`
d. `name-2`

2. Value types can be assigned Null

a. True
b. False

3. Which of the following is true?

a. Garbage Collector cleans up old memory no longer being used on the stack
b. Garbage Collector cleans up old memory no longer being used on the heap
c. Both of the above
d. None of the above

4. A derived class can be a base class.

a. True
b. False

5. Which of the following is true?

a. If a base class has virtual methods, you can override them in derived classes
b. If a base class has abstract methods, you can override them in derived classes
c. Both of the above
d. None of the above

6. Which of these pieces of code will compile? (Choose all that apply)

a.
```
var tmp = null;
```
b.
```
string x = null;
var tmp = x;
```
c.
```
var tmp = (string)null;
```
d.
```
var v = new object();
```

7. Which of the following statements is true? (Choose all that apply)

a. A Class can implement more than one interface
b. A Struct supports inheritance
c. A Class can implement more than one base class
d. A Class can be converted to Struct and vice-versa

8. Which of the following statements are true? (Choose all that apply)

a. `out` parameters must be initialized inside the called method
b. `out` parameters must be initialized inside the calling method
c. `ref` parameters must be initialized inside the called method
d. `ref` parameters must be initialized inside the calling method

9. Which of the following statements are true? (Choose all that apply)

a. All Operator overloads must be *public* and *static*
b. We can have multiple overloads of the same operator
c. You can overload the '&&' and '||' operators
d. Relational operators (eg: ==, !=) when overloaded, must return either a *char*, *int* or *bool*

10. What will the following piece of code return?

```
DateTime? dt = new DateTime();
dt = null;
Console.WriteLine(dt.ToString());
```

a. null
b. String.Empty()
c. The current date and time
d. The code will not compile

11. What is the value of 'x'?

```
public static void Main()
{
  var x = 1;
  var y = 2;
  var z = 3;
  x = (y==z) ? 4 : 5;
  Console.WriteLine(x);
}
```

a. 2
b. 3
c. 4
d. 5

12. What is the value of y?

```
public static void Main()
{
  int? x = null;
  int y = x ?? -100;
  Console.WriteLine(y);
}
```

a. null
b. 100
c. -100
d. 0 (zero)

13. The value of 'y' and 'z' in the following piece of code will be:

```
public static void Main()
{
  double x = 2.6;
  int y = Convert.ToInt32(x);
  int z = (int)x;

  Console.WriteLine(y);
  Console.WriteLine(z);
}
```

a. 2, 2
b. 3, 3
c. 2, 3
d. 3, 2

14. To read the value of a private field on a class, which BindingFlags will you use? (Choose all that apply)

a. Instance
b. Nonpublic
c. IgnoreReturn
d. Default

15. What is the output of the following statements?

```
public static void Main()
{
   Console.WriteLine(Math.Round(0.5));
   Console.WriteLine(Math.Round(1.5));
   Console.WriteLine(Math.Round(2.5));
}
```

a. 0, 2, 2
b. 0, 2, 3
c. 1, 2, 2
d. 1, 2 ,3

SUBMIT HERE

Some questions have more than one correct answer.

All those who complete the quiz first and with all correct answers will get a mention in the next edition, with a special mention to the one who completes it first!! Have fun.

About the Author

suprotim agarwal

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

Microsoft® Most Valuable Professional

THANK YOU

FOR THE 24th EDITION

@craigber

@shobankr

@yacoubmassad

@damirarh

@maheshdotnet

@sravi_kiran

@suprotimagarwal

@saffronstroke

WRITE FOR US