# Dynamisch und gefährlich?

## C# Dynamics in freier Wildbahn

von Tim Bourguignon

I've heard more than once that hell will simply break loose if somebody uses the C# keyword "dynamic". Another interpretation of the same rule says that god will kill a kitten for each usage... and if I'm not that scared of the first one... we're talking about damn cute kittens here! To be honest, since its introduction in the .NET Framework 4.0 in 2009, I tried to spare the lovely kittens and religiously avoided using dynamics... until last year. Since then, if those balls of fur really do get in trouble when I use dynamics ; well, I'm going to be in real trouble with the PETA very soon.

### Dynamics?

Let's backtrack a little bit for those who closed their eyes on some „new“ C# features in the last years. C# is a statically typed language, so what's that „dynamic“ I'm talking about?

C# is indeed a statically typed language. The language expresses type restrictions that can be verified and validated at compile time. For example, the following code does not compile:

```
string lang = "C#";
lang++;
```

The compiler notices that the "++" operator cannot be applied to operands of type 'string' and raises an error. Similarly, unless we define such a „ToUppper“ extension method, the following will not compile either:

```
int TheAnswer = 42;
TheAnswer.ToUpper();
```

The compiler cannot find the definition of a 'ToUpper' method accepting a first argument of type 'int'.

Now let's try replacing the 'string' and 'int' types in the examples above with 'dynamic'. You should notice the compiler instantly smiling back at you with a Pinacollada in his hand and no errors whatsoever to report.

With dynamic typing, the type-safety is determined at runtime. Thus, if the examples above compile, they will also spectacularly crash once you try to execute them...

## Why do we have a "dynamic" keyword in C# then?

Let's consider the following examples described by Scott Hanselman on his blog [1]:

```
Calculator calc = GetCalculator();
int sum = calc.Add(10, 20);
```

We define a calculator and call its „Add“ method. Nothing to notify here. But what about using the generic type 'object' instead of Calulator?

In this case, the compiler does not know that the "Add" method exists and raises an error. We have to use reflection in order to get to the member and invoke it at runtime. It works, but you'll have to admit that the syntax is a bit less clean:

```
object calc = GetCalculator();
Type calcType = calc.GetType();
object res = calcType.InvokeMember("Add",
        BindingFlags.InvokeMethod, null, new
object[]  { 10, 20 });
int sum = Convert.ToInt32(res);
```

On the other hand, the „Add“ Method is there, so if we could just call it, bypassing the compile time checks, we know the application would behave fine at runtime. This is exactly the promise of the dynamic:

```
dynamic calc = GetCalculator();
int sum = calc.Add(10, 20);
```

The compiler accepts anything that comes after a dynamic object as long as it respects the C# syntax and will compile it using dynamic mechanisms.

This concept of looking at the Type of an object via its Methods and Attributes instead of its Class is called „Duck-Typing“ and was first described by James Whitcomb Riley: „Wenn ich einen Vogel sehe, der wie eine Ente läuft, wie eine Ente schwimmt und wie eine Ente schnattert, dann nenne ich diesen Vogel eine Ente.“ [2]. Duck Typing is a key concept of all Dynamic-Languages and drastically raises the flexibility of the languages but also reduces its robustness due to the lack of compile time checks.

In addition to losing the compile time type checks, using dynamics comes with another drawback (some might argue that it's actually an advantage): the compiler cannot infer what methods and properties you can use, and thus prevents the usage of a code completion tool like IntelliSense [3].

## Dynamic Languages

The one scenario where this dynamic usage really shines is of-course the interoperability with dynamically typed languages like Python and Ruby via their .NET bridges IronPython [4] and IronRuby [5]. The dynamic keyword allows scripts written in these languages to access the CLR and thus the .NET plateform specific features.

Here's a very simple example of that bridge [6]. Let's assume we have the following Python script that adds two values together in a „script.py“ file:

```
def add(a, b):
    "add(a, b) -> returns a + b"
    return a + b
```

We can write the following C# code that will leverage the dynamic keyword and use the „Add“ function defined in the script:

```
var pythonRuntime = Python.CreateRuntime();
dynamic pythonScript =
            pythonRuntime.UseFile("script.py");
var result = pythonScript.add(100, 200)));
```

## What's the point?

I know what you're thinking: this is neither new nor really exciting so why bother writing an article about it?

And you would be right if it stopped there. Dynamics can be very useful in interoperability cases but the use of that naked 'dynamic' keyword doesn't make much sense at first sight outside those boundaries.

Users of ASP.NET MVC Framework [7] experienced a first use of that dynamic power under the disguise of the 'ViewBag'. And lately, some Frameworks or tools like NancyFx [8], Simple.Data [9] or Oak [10] started using dynamics publicly and extensively (poor little kittens). So there seems to be some cases where using those nasty dynamics can add something to the equation.

### Discoverability

Before we digg deeper in those usages and structures, let's talk API.

One major drawback that one can oppose to the dynamics is the lack of discoverability. Since I cannot use IntelliSense to know what I could use on the objects I get passed, I have to know beforehand what I can do.

Designing an API is all about making clear to the user what he/she can use and what the API provides. Using Interfaces and well defined methods is one of the best practices there and the nature of the dynamic world renders that plain impossible.

Telling the user what is available is then a challenge. The frameworks using dynamics either have to rely on a cristal clear API or be able to cope with anything. Keep that in mind in the following examples.

### DynamicObject

The System.Dynamic.DynamicObject is the base class for all the dynamic objects. As described previously, a DynamicObject exposes members at run time instead of at compile time. Among others, it has three important functions that you can override:

- TrySetMember
- TryGetMember is called when a member of a dynamic class is requested and no arguments are specified
- TryInvokeMember is called when a member of a dynamic class is requested with arguments

Combining those functions in a smart way is the key.

## ExpandoObject

The System.Dynamic.ExpandoObject represents an object whose members can be dynamically added and removed at run time.

Here's an example:

```
dynamic myExpando = new ExpandoObject();
myExpando.number = 10; //Add a property
myExpando.Increment = (Action)(() =>
        { myExpando.number++; }); //Add a function
Console.WriteLine(myExpando.number); //Prints 10
sampleObject.Increment(); //Call the function
Console.WriteLine(myExpando.number); //Prints 11
```

Instead of Number or Increment, I could have used any word here. Those properties and methods do not exist and are defined at runtime. Internally Expando uses a custom dictionary implementation to hold the dynamic properties you might add to your expandable object. The „keys“ would be the property names and those can then be accessed directly via *„object.<key>“* syntax.

Note how I used 'dynamic' to declare „myExpando“. If I had used "ExpandoObject" or even "var", the object would then be statically typed as an ExpandoObject and 'myExpando.number' would not compile anymore!

The ViewBag used in ASP.NET MVC is very similar in behavior to the ExpandoObject. As a matter of fact, ExpandoObjects are property-bags per definition for most of the developers.

Alexandra Rusina explained in a blog post [11] how ExpandoObjects can simplify access to XML structures. Basically when you have complex hierarchical objects, dealing with XML structures starts getting messy ; and the deeper the hierarchy, the uglier the code. Via an imbrication of ExpandoObjects, one can re-create an XML-like-graph in a more compact and easier to manipulate manner while still being able to use LINQ-to-Object mechanisms.

Even more interesting are the facts that the ExpandoObject also implements the *INotifyPropertyChanged* and can host events directly as properties. This way, the object becomes observable and can be used in interesting Databinding scenarios ; which

is more or less what NancyFx is doing (see the chapter on Nancy below).

### ElasticObject

The ElasticObject [12], is an OpenSource cousin of the ExpandoObject.

Where the ExpandoObject supports only one level of depth, the ElasticObject supports the direct creation of hierarchical dynamic data structure and dynamic collections and its direct conversion from and to XML. I must say it is a nice project, but I find the ElasticObject less useful than it seems.

### NancyFx

Nancy is a Micro-Webframework. Its goal is to provide a lightweight and low-ceremony way of serving web pages ; per opposition to ASP.NET MVC which tends to follow an out of the box *„I'm the only thing you'll ever need since I do everything and its contrary“* philosophy. Nancy is ready-to-go out of the box with the minimum viable feature-set and is to be extended when needed.

Nancy uses dynamics for databinding. When you define a „route“, Nancy will capture parameters and serve them up using an ExpandoObject-like object. For instance the route below will be activated for a HTTP-GET Request on the "/hello/bob" or "/hello/nancy" URIs. Nancy will then capture the "bob" and "nancy" parameters and pack them up in the dynamic dictionary represented by the variable "parameters".

```
Get["/hello/{name}"] = parameters => {
    return "Hello " + parameters.name;
};
```

This use of a dynamic is very smart. Nancy has many ways to capture parameters (query string, captured parameters & body of the request) and can capture any kind of data, thus packing them in a dynamic object drastically reduces the complexity of the binding the framework has to offer.

Furthermore, having the definition of the ".name" value and its usage beside one another drastically reduces the discoverability issues.

Still, it is not easy to see what's in the „parameters“ variable ; you'd better know it once designing the application. For instance if a query string was also passed "/hello/nancy?option=default", it is not easy to know that this "option" is also available.

Nancy also uses dynamics in the objects returned by the routes. Nancy uses an adapter pattern to handle those objects and pass them on to a specialized module that will take care of them. Using dynamics lifts the weight on having to create interfaces that all the adapters will have to implement and thus keep the flexibility at its maximum... with all the drawbacks it also brings if things go wrong.

### Oak's Gemini

Oak is an open-source framework that attempts to disrupt the process of Single Page Application (SPA) development in the .NET space. It tries to bridge the gap between C# and Javascript by many means and ease the general development livecycle imitating some Rails / node.js development tools.

Among others, Oak makes heavy use of a self-written component called 'Gemini' that brings some capabilities of dynamic type systems to C#. A Gemini object is able to create properties on the fly, define functions and supports introspection. All in all, it could be seen as a cherry-picked subset of the Dynamic/Expando/Elastic-Objects with introspection on top e.g. a way to know which properties are currently available on an object at runtime without directly using reflection.

```
dynamic gemini = new Gemini(new { FirstName =
"Jane", LastName = "Doe" });
gemini.MiddleInitial = "J";

gemini.RespondsTo("FirstName"); //this would return
true
gemini.RespondsTo("IDontExist"); //this would return
false

gemini.SayHello = new DynamicFunction(() =>
"Hello");
gemini.SayHello(); //calling method

IDictionary<string, object> members =
gemini.Hash(); //Get the members of the object
```

```
IEnumerable<string, object> methods =
gemini.Delegates(); //Get the Methods of the object
```

Again, it is for model-binding that Oak uses Gemini the most. For passing data in and out of his models from and toward the views. But also as a key component for accessing its databases, converting Json data and so on.

## Simple.Data

Simple.Data takes another approach. Making an extensive use of the TryGetMember function, it allows the on-the-fly definition of Query-Like structures to manipulate Databases using a syntax much closer to human-language than any other query mean.

In the following example, „database“ is a dynamic object and the

```
database.Customers.FindAllByCustomerId(3);

//The produced query: SELECT * FROM Customers
WHERE CustomerId = 3
```

The strength of Simple.Data is to combine objects, commands and data in a very readable way.

Most of Simple.Data's classes extend the DynamicObject and uses regexes to parse the called members in the „TryGetMember()“ functions.

The usage of dynamics allows the flexible concatenation of database related terms. Providing a relatively small set of commands (Find, FindAllByXXX, OrderByXXX, Select, WithXXX, Where etc.) Simple.Data keeps the discoverability issues to a minimum and manages to provide an extremely rich O/RM API without the hurdle of a mapping configuration and SQL statements.

## Fazit

We've (briefly) seen in this article various constructs and usages of the dynamic keyword that are currently used in the wild. From the DynamicDictionary to the ExpandoObject, ElasticObjects and GeminiObjects via their usages in XML manipulations, NancyFx, Oak and Simple.Data.

Beside the mental gymnastic that goes with it, the most interesting part behind those concepts is actually the API building part. What tools to I want to provide to my customers (e.g. developpers) for using what I produced? Which language should they talk? Should they talk something that's easy for me to process or something that's easy for them to express what they want? I clearly vote for the second take on this and think dynamics haven't shown us their full potential in this regard yet.

These were the examples I know of. If you happen to cross path with some other specimens in the wild, I'd be really happy to know about it. If not for my personal satisfaction, do it for the kittens!

## Referenzen


[1] http://www.hanselman.com/blog/C4AndTheDynamicKeywordWhirlwindTourAroundNET4AndVisualStudio2010Beta1.aspx

[2] http://de.wikipedia.org/wiki/Duck_Typing

[3] http://en.wikipedia.org/wiki/Intelli-sense

[4] http://ironpython.net/

[5] http://www.ironruby.net/

[6] http://www.codeproject.com/Articles/121374/Step-by-step-guidance-of-calling-Iron-Python-Funct

[7] http://www.asp.net/mvc

[8] http://nancyfx.org/

[9] https://github.com/markrendle/Simple.Data

[10] http://amirrajan.github.io/Oak/

[11] http://blogs.msdn.com/b/csharpfaq/archive/2009/10/01/dynamic-in-c-4-0-introducing-the-expandoobject.aspx

[12] http://elasticobject.codeplex.com/