The new dynamic language extensibility model for ASP.NET

By David Ebbo

Microsoft Corporation

Introduction

Ever since its original 1.0 version, ASP.NET has supported language extensibility. This means that a third-party compiler vendor can add support for using a new programming language for ASP.NET pages. This model has worked quite well, and has allowed many languages to be used with ASP.NET—languages from Microsoft (C#, Visual Basic, J#, and JScript) and from external vendors, such as implementations of Eiffel and COBOL.

However, the downside of the current extensibility model is that it primarily targets statically compiled languages like C#, and is not well adapted to dynamic languages like Python.

We are therefore introducing a new model for language extensibility. The new model aims to fill the lack of support for dynamic languages, and it enables dynamic languages to fit much more naturally into ASP.NET. Our initial implementation is focused on the IronPython language, but in the near future we will extend the model to work with any dynamic language.

Why Support Dynamic Languages?

I'll begin by explaining the reason behind wanting to support dynamic languages in ASP.NET. The last thing I want to do is start a debate about the pros and cons of static typing versus dynamic languages. Instead, I'll summarize the reason for doing this in a single word: *choice*. There are many good static languages like C#, and many good dynamic languages like IronPython, and in the end the choice of what to use comes down to personal preference and to the nature of the project you're working on.

Giving ASP.NET users the choice of languages was part of the design since our first version, and this is just another step in that direction. Unlike a number of other Web platforms that support only a single language, the ASP.NET team wants to enable users to choose the language that fits them best.

The CodeDOM Model

Before getting into the new dynamic model, let's start with a discussion of the existing ASP.NET language extensibility model: how it works, what makes it good, and what makes it inappropriate for dynamic languages.

The Basic Steps of Page Compilation

The ASP.NET compilation model is based on a powerful .NET Framework technology named the Code Document Object Model, or the CodeDOM for short. This model enables code to be written in a language-independent way. The basic steps for processing ASP.NET pages using the CodeDOM are these:

1. **Parsing**. ASP.NET first parses the contents of the page (that is, of an .aspx file), keeping track of all the constructs created in declarative markup, including server controls, code snippets, and static HTML.
2. **CodeDOM tree construction**. ASP.NET then creates a language-independent *CodeDOM tree*. The tree represents a class that will be derived from **System.Web.UI.Page**. At this point, there are no language-specific assumptions in the class representation (other than the user-written code snippets, which ASP.NET does not understand). You can think of this CodeDOM tree as a sort of pseudo-code representation of the class.
3. **Code generation**. ASP.NET then instantiates a specific CodeDOM provider, based on the language attribute found in the .aspx file (such as the attribute `language="C#"` in the **@ Page** directive.). ASP.NET asks this provider to generate the source code for the derived class from the CodeDOM tree using the target language. (For example, it might generate the source code in C#.) This generated source code contains the actual implementation of a type derived from **Page**.
4. **Compilation**. ASP.NET then asks the CodeDOM provider to compile the source code into a .NET Framework assembly (a DLL). In most cases, the provider performs this step by launching a command line compiler, such as csc.exe for C# source code.
5. **Execution**. Finally, ASP.NET loads the generated assembly, instantiates the generated class, and uses it to execute the HTTP request.

Note that steps 1 through 4 happen only once, as long as the page doesn't change. Step 5 occurs for every request. An almost identical sequence occurs with user controls (.ascx files) and master pages (.master files).

> **Note** With the code-behind model in ASP.NET 2.0, there is also a user-written partial class that comes into the picture. However, this does not substantially change the structure of the generated class.

Code Generation for Declarative Controls and Markup

What happens to all the server controls, the HTML markup, and the code snippets in your .aspx file? They're all handled by code generated inside the derived class. ASP.NET generates code that builds the control tree, code that renders markup, and code for additional tasks like data binding. The code generation process is relatively complex; the important thing to understand is that page execution is driven by code.

Let's take a simple case where you have a **TextBox** control on your page, which might look like this:

```
<asp:textbox runat="server" id="MyTextBox" text="Hello" />
```

Somewhere in the generated code, and assuming that the page language is C#, there will be code that looks like the following, which builds and initializes the control:

```
TextBox MyTextBox = new TextBox();

MyTextBox.ID = "MyTextBox";

MyTextBox.Text = "Hello";
```

If you are interested in finding out more about the derived **Page** class generated by ASP.NET, I encourage you to take a closer look at it. Although it might not all make sense, you'll still recognize code that relates to many elements in your .aspx page, and you might find the code enlightening.

The simplest way to look at the generated code is to set `debug="true"` in the **@ Page** directive of the .aspx page. Then purposely introduce a syntax error in a server script block (a **<script>** element with the attribute `runat="server"`—for example, just create a line that says `SYNTAX ERROR`!. When you request the page, you will see an error message that includes a `Show Complete Compilation Source` link. Click the link and you will see all the generated code. Look for the IDs of some of your server controls, and you will see how the controls are built and added to the tree.

What Makes the CodeDOM Powerful

The CodeDOM provides a powerful layer of abstraction between ASP.NET and the programming languages used to create page logic. This abstraction enables ASP.NET to support an arbitrary set of languages without having any knowledge about them. And it goes the other way as well—the implementer of a CodeDOM provider does not need to know anything about ASP.NET. In fact, a CodeDOM implementation is useful in many scenarios that have nothing to do with ASP.NET.

This is a much better model than one where ASP.NET would be hard-coded to work with a small set of Microsoft languages, and users would need to wait for a new version of the .NET Framework to get expanded language support.

The CodeDOM and Dynamic Languages

If the CodeDOM is so great, why are we coming up with a new one?

The answer is that even though it is language independent, the CodeDOM does make a number of assumptions about the capabilities of supported languages. In particular, it assumes that any language used for ASP.NET has the ability to produce true classes in the .NET Framework sense—that is, classes that are in on-disk assemblies and that can be loaded using standard APIs like **Type.GetType**. Those classes must be able to inherit from other classes like **System.Web.UI.Page**, override base class methods, and declare methods with very specific signatures.

Unfortunately, for most dynamic languages, these seemingly simple requirements are essentially out of reach. Even though dynamic languages might have some form of class construct (as in Python), this capability does not easily map to the .NET Framework-style classes that we need, mostly because of the lack of strong typing. For example, in C# you can write a method that takes a string and returns an integer; in IronPython, you have no way of specifying such a typed signature. In addition, inheriting from existing classes and overriding specific methods in IronPython is more difficult.

Because ASP.NET is designed to inherit from classes like **System.Web.UI.Page** and to override a number of methods, we were faced with an interesting challenge when we decided to add IronPython support to ASP.NET.

In fact, we initially experimented with using the CodeDOM approach. We wrote a prototype CodeDOM provider for IronPython, and we had some success getting it to work with ASP.NET in constrained scenarios. But we eventually realized that making the CodeDOM work fully with IronPython would require extending the language (for example, to add ways to specify typing). We felt that this was not the right direction. Also, writing a CodeDOM provider is a non-trivial task, so requiring each dynamic language to provide one would make it harder for many languages to adopt the model and support ASP.NET.

At this point where we went back to the drawing board and decided to design a new extensibility model that makes more sense for dynamic languages in ASP.NET.

"No-Compile" Pages

A short detour: need to explain a little-known feature that is already part of ASP.NET 2.0: so-called *no-compile* pages. This is important because the new model is based on this feature, and then extends it to support dynamic languages.

This feature is triggered by the **CompilationMode** attribute in the page directive, which might look like the following:

```
<%@ Page CompilationMode="Never" %>
```

The no-compile option is also used if you set the **CompilationMode** attribute to "Auto" in a page that has no code.

As the name indicates, a no-compile page is not compiled. In contrast with compiled pages, which are code driven, no-compile pages are entirely data driven, and require no compilation at all. As a result of not being compiled, they are faster to process and more scalable (more on this later).

So what's the catch? Well, it's a big one: those pages cannot contain any user code! Instead, they're limited to static HTML and server controls. Obviously, this is a big limitation, which explains why no-compile pages are not widely used. But as you'll see later, our new model removes this restriction and enables dynamic code in no-compile pages. Let's spend a little more time discussing how no-compile pages work by comparing the behavior of compiled and no-compilation pages. Again, I can use .aspx pages to illustrate the concept, but things work the same way with user controls and master pages.

How No-Compile Pages Work

All I've said so far is that no-compile pages are data driven instead of code driven, but what exactly does that mean? If you recall the basic steps I described for the compiled CodeDOM, it starts with parsing the page. The parsing step also happens for no-compile pages, but that's the only step they have in common; everything else happens very differently. Let's look at those steps in more detail:

1. **Parsing**. ASP.NET first parses the contents of the page and keeps track of all the constructs such as server controls, code snippets, and static HTML.
2. **Control builder tree construction**. From the parse information, ASP.NET creates a special data structure that keeps track of everything that it needs to know in order to create pages. Let's use the same <asp:textbox> example we looked at earlier. Instead of a resulting in a piece of generated code, the text box becomes a node (called a *control builder*) that knows how to create this **TextBox** control and initialize it with all the attributes that you set.
3. **Execution**. When the page needs to be instantiated for execution, ASP.NET asks the control builder tree to instantiate it, along with all its controls.

In the no-compile scenario, steps 1 and 2 only happen once (as long as the page doesn't change), while step 3 happens on every request.

In this model, no class is ever derived from the base **Page** class. Instead, the **System.Web.UI.Page** class (or optionally a custom base page if there is an **inherits** attribute) is instantiated directly.

What Good Are No-Compile Pages?

In spite of their no-code limitation, no-compile pages are far from useless. If you have a set of controls that encapsulate their own functionality (such as a weather widget), you can put together useful pages without any code. In fact, the next major version of SharePoint relies heavily on this feature, for at least two good reasons:

- The SharePoint team doesn't *want* user code in certain pages, for security reasons.
- No-compile pages give SharePoint better scalability because there is no compilation.

But I would certainly agree that for the general ASP.NET developer, no-compile pages are of limited use. The new model that we are creating for dynamic languages will likely change this, because it allows no-compile pages to have code.

You now have enough background information that we can start looking at this new model.

The New ASP.NET Model for Dynamic Languages

***Disclaimer**: We are still at a very early stage in this project, so details are subject to change. In addition, we have not yet reached the point where we have a generic pluggable model, because at this time we are supporting only the IronPython language. As a result, the discussion that follows is limited to a high-level description of how the model works, and does not explain how you could integrate new languages into this model. But the time for this will come soon!*

Up to this point, I've looked at two models for ASP.NET pages—one that allows code but requires static compilation, and one that does not require any compilation but doesn't allow any code. To reach our goal of integrating dynamic languages into ASP.NET, we need a hybrid model that doesn't cause static compilation but still allows the page to contain code.

I'm not saying that no compilation at all should ever occur in the new model. In fact, dynamic languages have a lot to gain from compilation in terms of performance. But what we wanted to avoid was the requirement of *static* compilation, by which we mean the generation of an on-disk assembly implementing standard .NET Framework types.

How the New Model Integrates into ASP.NET

For the most part, the new model builds on top of ASP.NET 2.0. However, we needed to make a small change to the ASP.NET parser. This section describes the change, as well as the various ASP.NET extensibility features that the new model uses to integrate into ASP.NET.

As I've discussed, the new model is based on the ASP.NET no-compile feature, and as a result it works very similarly to what I described above. However, the problem with no-compile pages is that normally the parser fails instantly when it finds any code in the page. Obviously, this is a problem for us if we are going to support dynamic code!

Even though we were really hoping to implement the new model without changing System.Web.dll (the main ASP.NET assembly) we found that we needed some small changes to the parser to enable it to accept code in no-compile pages. For that reason, when you install the dynamic language support, you get a new version of System.Web.dll.

The change to ASP.NET is in the **PageParserFilter** API, which gives external code a hook into the parser. This **PageParserFilter** API already existed in ASP.NET 2.0; it was simply expanded to accommodate the new model.

Custom PageParserFilter

In the new model, we register a **PageParserFilter** class in order to customize the parsing behavior and allow code in no-compile pages. A Web.config file for a dynamic language application will include this element:

```
<pages
pageParserFilterType="Microsoft.Web.IronPython.UI.NoCompileCodePageParserF
ilter"
... />
```

The **PageParserFilter** class does the following:

- If a page uses a dynamic language (currently, only IronPython is supported), it makes sure that the page inherits from the proper base class. (More on this base class later.)
- If the parser encounters a code snippet (`<% ... %>`, `<%= ... %>`, or `<%# ... %>`), it replaces the snippet with a special control that holds the code. From the point of view of ASP.NET, there is no code; the snipper just becomes a control that happens to have a string property with the code. This is how we get around the restriction that no-compile pages can't have code.
- If the parser finds an event hookup attribute (such as `onClick="`*`MethodName`*`"`), it replaces the attribute with a special control, much as with code snippets.
- If the parser finds a `<script runat="server">` element, it handles the element as a page-level property, which later is passed as a string to the custom base class.

Custom HTTP Module

In the new model, we also implemented a custom HTTP module (that is, a class that implements **System.Web.IHttpModule**). You can see this registration in the Web.config for an IronPython application:

```
<httpModules>
  <add name="PythonModule"
type="Microsoft.Web.IronPython.DynamicLanguageHttpModule"/>
</httpModules>
```

The HTTP module is used to hook early into the application domain cycle and register various components with the dynamic-language environment. (The application domain is an area within the process where all the code from one Web application is executed.) The module is also used to implement an equivalent to the Global.asax file for dynamic languages, which will be discussed later.

Custom Base Class

In order to implement its behavior, the new model relies on having all pages that use dynamic language extend a special base class named **ScriptPage**, which in turn extends **System.Web.UI.Page**. Similarly, we have special base classes for user controls (**ScriptUserControl**) and master pages **(ScriptMaster**). Having these base classes gives dynamic language pages a way to participate in the page life cycle and make everything fit together.

ASP.NET Features Supported in the New model

Let's pause here and look at the features that the new model supports.

Standard Page Functionality

Dynamic language pages don't look much different from regular ASP.NET pages, and can essentially use all standard ASP.NET features, including the following:

- Pages (.aspx), user controls (.ascx), and master pages (.master) are all supported.
- Pages can contain server controls—that is, elements with the `runat="server"` attribute.
- Pages can use the in-line code model or the code-behind model. As you will see, this works a bit differently than standard compiled pages, but it accomplishes the same thing.
- Pages support code snippets (`<% ... %>`, `<%= ... %>`, and `<%# ... %>`). Again, they work a little differently, but not all that much.

If you've used standard ASP.NET pages, there should be a very short learning curve for using dynamic-language pages.

Application File

The new model supports a file similar to the Global.asax file, but it works a bit differently. Instead of Global.asax, the file is named Global.*ext*, where *ext* is the language-specific extension (for example, for IronPython the file name is Global.py).

One important difference is that this file contains only code, unlike Global.asax, which contains a directive (`<%@ %>` element) and a script block with a `runat="server"` attribute. For example, a simple Global.py might contain the following:

```
def Application_BeginRequest(app):

    app.Response.Write("Hello application!");
```

App_Script Directory

A dynamic language application contains an App_Script folder that is similar to the App_Code directory, except that it contains dynamic-language script files instead of static language code files. But the general idea is the same: files in this directory contain classes that are usable by code anywhere in the application.

Generic HTTP Handlers

A dynamic language application can contain an HTTP handler that is the equivalent of an .ashx file in a standard ASP.NET application, but again it works a bit differently. As with the Global.py file, handlers contain only code, unlike .ashx files, which contain a directive that ASP.NET recognizes. Dynamic language handlers are named using the pattern Web_*name*.*ext*. In IronPython, *.ext* is .py—for example, Web_MyHelloHandler.py). The Web_ prefix is significant, because it is registered to specify that the code file is an HTTP handler.

Dynamic language handlers must contain a **ProcessRequest** method, which is invoked to handle the HTTP request. This is very similar to **IHttpHandler.ProcessRequest** in .ashx files. For example, a dynamic language handler file might contain the following:

```
def ProcessRequest(context):

    context.Response.Write("Hello web handler!")
```

Not Supported: Web Services

The new model does not currently support an equivalent of .asmx Web services. The Web service architecture works only with standard .NET Framework types, which as noted earlier are difficult to create with dynamic languages. To make things even trickier, Web service class methods must be decorated with special metadata attributes (like `[WebMethod()]`), and dynamic languages typically have no syntax to do this.

We are hoping to come up with a solution for this limitation.

How Code Is Handled in the New Model

As you have seen, dynamic language pages are based on no-compile pages, but can nonetheless contain code. In this section I'll show you how this code is actually handled.

In contrast to the CodeDOM, where all the user code in a page becomes part of a generated source file, in the new model each piece of user code in a page is treated as an individual entity. Let’s look at the various types of user code to understand how they are used.

Code in <script> Elements

In standard ASP.NET pages, user code in **<script>** elements with a `runat="server"` attribute ends up inside the body of the generated class, which is why it can contain method and property definitions as well as field declarations. But in dynamic language pages, we don't generate a new class at all; instead, ASP.NET directly instantiates the class specified by the **inherits** attribute (the **ScriptPage** class). In this respect, the term "inherits" is inaccurate for dynamic language pages (and for no-compile pages in general), because there is no inheritance occurring.

What happens to the contents of the **<script>** element? Instead of becoming part of a class, the code becomes a kind of companion code for the **ScriptPage** class; you can also think of it as a pseudo-partial-class. Nomenclature aside, let me explain how it works by using a simple IronPython example like this:

```
<script runat="server">

def Page_Load():

    Response.Write("<p>Page_Load!</p>")

</script>
```

Here, the **Page_Load** method is not part of any class. Instead, members of the page class (typically of type **ScriptPage**) are 'injected' by ASP.NET in order to be directly available (hence we're able to use 'Response' directly). So for all practical purpose, you can think of your methods as being part of the page class, even though from a pure Python perspective they are not part of a class at all.

In IronPython terminology, the code in the script block lives in a *module*. Generally, there is one module associated with each page, user control, or master page. (Note that there is only one module instance per page, not an instance per HTTP request.)

Let's look at a different example:

```
<script runat="server">

    someVar=5

</script>
```

Here, it is important to understand what the scope of the `someVar` variable is. Given that it is a module level variable (module in the IronPython sense), and that there is only one module instance per page, it follows that there is only one instance of the variable. So semantically, it is very similar to having a static field in a regular ASP.NET page.

Code-Behind Files

The new model supports putting code in a code-behind file, as with the standard ASP.NET model. However, there are important differences in how this works in the two models.

In the standard model, the code-behind file contains a partial class declaration, which is merged by the compiler with the generated class.

In the new model, there is no class declaration in the code-behind file. Instead, methods appear directly in the file, outside of any containing construct. If this sounds similar to what I described for in-line code (code in `<script runat="server">` blocks) in the preceding section, it's because there really is no difference. In the new model, you can take the exact content of a `<script runat="server">` element and move it to a code-behind file (for example, .g. MyPage.aspx.py if the page is MyPage.aspx) without changes.

Thus, everything I discussed above about the scope of variables applies equally to code-behind files.

As with normal ASP.Net files, whether to put your IronPython code in line or in a code-behind file is purely a matter of personal preference: do you prefer to see the code directly in the .aspx file, or would you rather keep it in its own code file? The choice is yours.

Code Snippets

Snippet expressions (`<%= ... %>`) and statements (`<% ... %>`) also execute in the context of the module created for the page's code. As a result, these snippets have access to methods and variables defined in with in-line or code-behind code. For instance, if you define a `Multiply` method in the **<script>** block, you can write `<%= Multiply(6,7) %>` in your page.

Code snippets also have access the members of the Page class. For example, you could write `<%= Title %>` to display the title of the page.

Data-binding Expressions

Even though data-binding expressions are a type of code snippet, they deserve being discussed separately. The reason they are particularly interesting is that in IronPython, data-binding expressions work more naturally than they do in standard pages.

If you have used ASP.NET data-binding expressions, you are likely familiar with the **Eval** method. For instance, you might have a **GridView** control with a templated column containing the data-binding expression `<%# Eval("City") %>`. Here, the **Eval** method is used to get the value the column named `City` for the current row in the database (or for whatever data source you are using). This works, but the fact that you must go through this **Eval** method is rather awkward.

In the new model, the equivalent is to simply use the snippet `<%# City %>`. Here, `City` is an actual code expression in the dynamic language, instead of a literal string that must be interpreted by the **Eval** method. Hence, you are free to write arbitrary code in the expression. For example, with IronPython you could write `<%# City.lower() %>` to display the `City` value in lower case. This improved syntax is made possible by the late-bound evaluation supported in dynamic languages. Even though the meaning of `City` is not known at parse time, the dynamic language engine is able to bind it to the correct object at run time.

Dynamic Injector Mechanism

Another case that demonstrates the flexibility of dynamic languages over static languages is the injector mechanism supported by the new model. This is best demonstrated using an example.

Imagine that you have code in a page that reads a value from the query string. The page URL might look like this:

```
http://someserver/somepage.aspx?MyValue=17
```

In a C# page, you would get the value using the code like the following:

```
String myValue = Request.QueryString["MyValue"];
```

But in a dynamic language application you can simply write the following to achieve the same thing:

```
myVar = Request.MyValue
```

What exactly makes this work? In the new model we register a special object known as an *injector*, which says something like the following to the dynamic engine: "If you find an expression *SomeObj.SomeName*, where *SomeObj* is an **HttpRequest** object and *SomeName* is not a real property of the **HttpRequest** object, let me handle it instead of failing."

The way that the injector handles the expression is by calling *SomeObj*`.QueryString["SomeName"]`. Even though the expression `Request.MyValue` looks simpler than `Request.QueryString["MyValue"]`, in the end it is really executing the same logic.

The same injector mechanism is also useful in other cases. For example, where you would write `SomeControl.FindControl("SomeChildControl")` in C#, you can simply write `SomeControl.SomeChildControl` in a dynamic language application.

Furthermore, the injector mechanism is extensible, so if you have your own type of collection that is indexed by string, you can write a custom injector for it to simplify the syntax.

Though they may not be revolutionary, features like this and like the simplified data-binding expression contribute to making life easier when writing Web applications.

Compilation of Dynamic Code

I have discussed the fact that the new model is using the no-compile feature of ASP.NET. This may lead you to believe that the code in dynamic language pages is interpreted, but that is not the case. Though this may appear to contradict the definition of the no-compile feature, the code in dynamic language applications *is* compiled.

The explanation is that the term "no-compile" refers explicitly to CodeDOM-style static compilation, which is not used in the new model. But this does not prevent dynamic code from being compiled on the fly by the dynamic language engine, and that is what we do. As you would expect, the benefit of compiling the code is that it executes much faster than if it were interpreted.

Conclusion

In this document I have looked at both the existing model and the new model for integrating programming languages into ASP.NET. It is not my intention to convince our Web developers to stop using C# and switch to IronPython. Rather, my goal is simply to explain the differences between the two models, and to keep you aware of what we are working on. As mentioned, this is still very much work in progress, so you should expect a revised version of this paper in the future.