Integrating Apache with .NET

A Presentation for…
ApacheCon 2002, Las Vegas
By Derek Ferguson

Good morning and thank you for coming to my session today. The name of this session is "Integrating Apache with .NET". My name is Derek Ferguson, and I am the Chief Technology Evangelist for Expand Beyond Corporation, the worldwide leader in mobile software for enterprise management.

I've divided today's presentation into three main categories. To begin with, I'm going to explain to you a little bit about what exactly .NET is and what it can do for you. Then, in general terms, I'm going to look at how you can get .NET to interoperate with pretty much any technology that runs on the Java platform. Finally, I'm going to "zero in" on Apache and give you some very exact guidelines for how to use Apache with .NET.

Now, the first question that you might have is, "Who am I to be telling you all of this?" Well, in the .NET world, probably my best credential is the fact that I was one of the first 50 non-Microsoft people in the world to see .NET up-close-and-personal at a briefing for Microsoft technology authors in Redmond way back in May of 2000. Back in those days, it was still known as COM+ 2.0 – a name which lacked the marketing ring of .NET but which I think was at least a little more descriptive for most developers.

Since then, I've gone on to write one of the most widely read books on .NET mobility, which is called (appropriately enough) *Mobile .NET* (APress, 2001). And I also serve as Editor-in-Chief for Sys-Con Media's newest print publication, the *.NET Developers Journal*. Sys-Con Media, for those of you who might not immediately recognize the name, is the publishing company behind *Java Developers Journal* – the most successful Java print magazine out there.

So, that kind of leads me into the whole Java area of my credentials. The fact that I work with Java at all certainly puts me in the vast minority of .NET developers. The fact is, though, I very much like Java and use it very extensively in my day job with Expand Beyond. As you'll see in my next presentation, for example, our entire line of products is actually built using Apache Jakarta technologies.

In terms of Java credentials, to begin with I have spoken at Java One, which is Sun's big annual Java conference – so certainly a kind of endorsement from the people who created Java in the first place. And also, as I just mentioned, I work with Sys-Con, which has a reputation for being just about the only publisher in the industry that is willing and able to do both the .NET and Java sides of the technology universe. And, finally, I've written articles for O'Reilly's OnJava.com web site.

The final thing that I'd like to say about myself before we proceed is that I strongly encourage you to contact me with any questions or comments whatsoever at derek@XB.com. Really, I'm very serious about this – no one ever contacts me about anything after these presentations and I'm really quite excited to hear what you all are working on and to be of assistance in any way that I can.

Introduction to .NET

So, the first thing to understand about .NET is that there are really three parts to it. The first part is the marketing hype. This refers to all of the "1 Degree of Separation" ads that are out there which tell you absolutely **nothing** about what .NET is and can do for you as a developer. It also refers to Microsoft's practice of suddenly attaching the suffix ".NET" to everything they produce, whether it actually has any technological relationship to the .NET platform or not. Needless to say, we won't be discussing any of these aspects of .NET in this presentation.

The second part of .NET is Microsoft's vision of selling software as a service, rather than as a shrink-wrapped commodity for which you only have to pay once. The infrastructure which Microsoft ultimately sees supporting this vision is the Web Services platform supported by .NET and other XML-based application servers. We will indirectly address this point in this presentation simply by virtue of the fact that we will be looking at how to use Web Services to help us achieve interoperability between Apache and .NET.

The final part of .NET, though, is the part that is nearest and dearest to our hearts as developers, and that is the actual development platform offered by .NET. In short, .NET represents Microsoft's embracing of the "virtual machine" model for enterprise software creation and deployment. What this means is that, under .NET, developers no longer write code which is ever compiled to native code, in contrast to the way things were under Microsoft's previous platform – COM – in which everything occurred right at the OS level.

Microsoft's virtual machine is not the JVM, however, it is, instead known as the Common Language Runtime (CLR). Furthermore, this virtual machine does not run Java byte codes; instead it runs byte codes from the Common Intermediate Language (CIL). Often, you hear this referred to as MSIL, which means Microsoft Intermediate Language. It is not intended to be 100% platform independent like Java, but – insofar as it avoids heavy linkage into the operating system – it is easily ported to other platforms. In fact, Microsoft themselves co-funded a project with Corel that ported large portions of .NET to the FreeBSD operating system.

.NET derives two main advantages from the virtual machine software model. To begin with, developers no longer have to worry about corrupting memory and creating errors like General Protection Faults. .NET software may not function, but – like Java – it typically doesn't "blow up" and take resources with it. If it does, you can open a trouble ticket with Microsoft and have them fix their infrastructure – which is a lot cheaper and quicker than trying to debug your own code… hopefully.

The other benefit is that, since the CLR is constantly interpreting every instruction that needs to be executed, it really has the power of life-and-death over all of your code. This means that you can have much finer-grained control over code security than was ever possible with COM. This also means that the CLR is capable of detaching and reattaching code from the file system while it operates – allowing you to “hot deploy” new code to systems while they are running. For example, you can upload a .NET assembly to a web site and have it take effect immediately without even having to restart the server.

In addition to this new virtual machine, Microsoft has created a huge library of code components known as the .NET Framework. The idea here is to give the developer as much pre-built code as is available to developers on other platforms, such as Java. Making the .NET Framework a part of the basic .NET distribution – which is also completely free, for those of you who didn’t know – has two basic effects:

1. The time to create custom applications is greatly decreased
2. There is much greater consistency in how common tasks are performed across different organizations

In addition to the virtual machine and the Framework, Microsoft offers Visual Studio .NET. Now, unlike in the previous, COM generation of Microsoft tools – you don’t need Visual Studio .NET in order to create .NET applications. If you prefer to create your code using Notepad or some other text editor and then compile it with the completely free command-line compilers that are included with the .NET Framework SDK, then go right ahead. Visual Studio .NET just adds an IDE and many ease-of-use features.

There are five basic kinds of .NET application:

1. Web Applications – applications which run via the World Wide Web, using exactly the same application programming model as regular, Windows-based applications.
2. Web Services – XML-based software components that can receive procedure calls via SOAP, HTTP, and other open Internet protocols
3. Windows Applications – standard, desktop-based applications
4. Mobile Web Applications – Web applications which use an abstract control set for page construction and which are then able to dynamically adapt their output to be either XHTML, WML, or cHTML
5. Smart Device Applications – Applications which actually run on Windows CE-based devices

.NET and Java Conversion

Now, the most general kind of interoperability between .NET and Apache will take place at the .NET-to-Java layer. The thinking here is that you have Apache technologies which are running on the Java platform and other applications running on the .NET platform. So, the most basic way to get them to interoperate is to get their foundation technologies started talking to one another.

Microsoft's primary suggestion for people who have existing Java technologies that they want to work with .NET seems to be:

"*Why not port them to .NET?*"

Well, we'll leave the matter of whether or not this is always a good idea in abeyance for the moment and consider the technologies that are available to assist in this pursuit. Basically, there are two paths you can take.

The first path will take your existing Java code and move it completely over into ".NET land." The distinguishing characteristic of this path is that you lose all access to both the Java language and platform when you go down it. For example, you will have to only use the .NET Framework from now on, rather than any of the packages in the Java libraries.

There are two tools that will take you in this direction. The first of these is known as the Java Language Conversion Assistant and is currently in Beta 2 and available for free downloading from MSDN (URLs at the end of this paper). This tool expects you to provide it with Java source code, which it will then try to convert into C# source code that references .NET libraries instead of Java libraries. When it is done, you will be provided with the source code and a report of how well the conversion attempt seemed to go.

Typically, this tool achieves about an 85% success rate on conversions. If you have very simple code, it should be sufficient. If you have more complicated code, you will either have to do a great deal of additional conversion work, or you might buy the commercial version of the tool from the company that produced the free version for Microsoft – ArtinSoft.

One major problem with the JLCA is that it requires source code in order to work. Sometimes, you might want to use a piece of Java code – an encryption library, for example – for which you don't have the source code. For cases like this, Microsoft has created a tool called JbImp, which will actually try to make a bytecode-to-bytecode conversion from Java to .NET.

The good news is that, like the JLCA, JbImp has about an 85% success rating. The unfortunate part is that, if it fails even 1% -- what do you do? Presumably, you are using JbImp because you don't have the source code for the Java you are trying to convert. So, it is very much an all-or-nothing proposition, most of the time.

The other approach to converting Java for use under .NET is to write your code for J#, which is:

- Java language syntax
- Java libraries circa v1.1.8
- .NET libraries

In many cases, this might be the best route. For example, if you have a group of Java developers who need to create some .NET code, they might find using J# easier than jumping right into C# or VB.NET. This way, they have most of their existing libraries, and only have to learn a few .NET libraries to "make up the difference."

On the other hand, J# is **not at all** compatible with pure Java. You cannot access real Java components from J#. This means, for example, that JDBC data access – which would require the use of a pure JDBC driver – is impossible… unless, of course you could find some way to trick .NET into interoperating with pure Java.

.NET Consuming Java classes

There are three ways to allow .NET to consume Java classes without resorting to the overhead of Web Services. The first of these is now considered legacy technology and is called the JavaBeans Bridge for ActiveX. The second of these is still in "Early Access" and is known as the J2EE CAS COM Bridge. The final way requires spending some money, as it is a commercial solution, JNBridge.

In order to understand either of the first two solutions, you must understand that .NET ships with a "COM interoperability layer." This allows .NET to do two things:

1. Make any of its own components "look like" COM components, to allow them to be consumed by any legacy COM applications
2. Consume any COM component, by placing a ".NET wrapper" around it

What this means to you is that any technology that can make Java classes "look like" COM components can be consumed by .NET via .NET's COM Interoperability layer. The obvious trade-off, however, is two levels of translation between your .NET code and the Java code you wish to access. This will result in **some** kind of performance hit – but not nearly what you'd have with the typical Web Service.

The JavaBeans Bridge was the initial solution proposed by Sun Microsystems for allowing Java technologies to be accessed from "legacy" COM clients. The technology basically takes any Java Bean and wraps it in a COM component interface that is capable of running the Java Bean through a connection into whatever JVM is installed on the machine. This is an excellent solution, but unfortunately, it only works for Java Beans – not for any of the other myriad kinds of Java classes (not even for Enterprise Java Beans… in case you were wondering).

If you want to access a kind of Java class that **isn't** a Java Bean, then, you have two choices. The first of these is to wrap up access to whatever kind of Java class you actually want to access in a custom JavaBean that exposes its functionality. The other alternative is simply to choose another interoperability technology.

The J2EE CAS COM Bridge is an excellent choice as a technology for allowing .NET to access Java classes. This technology is also produced by Sun Microsystems and is 100% free for downloading. It also works with any kind of Java class – even EJB's running on remote computers under some kind of J2EE application server!

The only downside I have found to the J2EE CAS COM Bridge is the fact that it is still in Early Access. If this presents operational road-blocks at your organization, you might have to wait for it to go G.A. Or, you might consider using a commercial solution such as JNBridge.

JNBridge is a very clever solution to the problem of Java interoperability. Essentially, what they have done is to create a "pure Java" server that is capable of speaking the .NET Remoting protocol. .NET Remoting is Microsoft's successor to DCOM – the binary-level (as opposed to XML… which would take us back into the whole "Web Services" realm) protocol used by .NET components to talk to each other across process – and even network – boundaries.

Now, in actuality, .NET Remoting can operate either using binary or XML messages and either TCP or HTTP protocols. But, for purposes of our conversation, we're going to stick to binary/TCP – because this is where you get your real performance savings over Web Services. So, by creating this Java server that can speak .NET's native binary/tcp protocol for remote procedure invocations, JNBridge took the first step towards allowing completely seamless, high-performance access to pure Java code from .NET.

The other important piece to this solution, however, is a GUI that can load in any Java class (plus, automatically, all of its dependencies) and create .NET proxies for them which automatically communicate with their server. This way, all of the details of .NET Remoting are completely concealed from you as a developer. To your perspective, it just looks as if Java classes have suddenly become available right from the .NET References portion of the Visual Studio IDE!

The only downside to JNBridge that I have found is the price tag. It costs \$500-\$2500 per developer at design time and \$100-\$1000 per unit shipped at runtime. These costs, I would think, make it a certainty that this technology will only ever be used internally – never as a part of any commercially shipping solution.

.NET and Apache ANT

So, at this point, you have learned enough techniques to allow you to access virtually any piece of Java code directly from .NET without having to resort to Web Services. There are just two problems remaining:

1. How do we access .NET code from Java?
2. How does any of this apply directly to Apache?

The simple answer to the first question is “Web Services.” There are many technologies available for building Web Service clients using Java, and I will tell you about the one that I think is the easiest – so you can get up-and-running the quickest.

So far as Apache itself is concerned, however, most of the techniques described already will apply directly to Apache – particularly the Jakarta project, with which I am most directly familiar. However, there are a couple pieces of Apache – ANT and the native web server, for example – which offer additional opportunities for .NET integration, which have not yet been discussed.

ANT, in case you are not already aware, is Apache’s answer to the UNIX *make* tool – which was also the basis for Microsoft’s legacy *NMAKE* tool. The problem with *make* was always that it was far too UNIX-and-C-centric – it never gave you the feeling that it was really intended for use with either Java or .NET.

The problem with Microsoft’s tools, on the other hand, was always that they were really intended for use strictly through the IDE. Whatever functions were put into *NMAKE* were always a very small subset of the features available through IDE-based compilation, at best. Before .NET, Visual Basic code couldn’t be compiled from the command line **at all**!

Now that .NET supports full command-line compilation for C# and VB.NET, one of the barriers to production-level building of Microsoft platform code (automated nightly builds, for example) has been removed. ANT has added some .NET specific tags to help this process continue to its natural conclusion:

- CSC is a tag for running the .NET C# compiler
- ILASM is a tag for assembling IL files
- WSDLTODOTNET is a tag for access the WSDL tool

Web Services

When I initially proposed this session two years ago, Web Services were going to form the vast majority of my talk. Since then, however, Web Services have lost a lot of their “cutting edge” feel – so I don’t talk about them quite as much. Part of the reason for this is that people are simply more familiar with them now. The other part of the reason for this is that they simply have **not** yet taken off the way that everyone has hoped that they would.

Part of the reason that they haven’t take off is certainly the economy. People simply don’t want to invest money in large, experimental Web Service systems that are intended to connect them to partners – who may or mayn’t be interested in even consuming their Web Services. They’re even less willing to build consumer-oriented Web Services, given the general lack of consumer interest in additional technologies since the Internet boom faded two years ago.

However, the one bright spot for Web Services lies in Enterprise Application Integration – so that is exactly what we can use them for in this case. If you have some .NET code and you want to access it from some Java code – Apache Tomcat, for example – then the best way to do this at present is to wrap your .NET code in a Web Service and access it via XML using either the Sun Java Web Services Developers Pack (JWSDP) or GLUE.

Sun's Java Web Services Developers Pack has the distinct advantage of being from Sun and, therefore, having the blessing of the creators of Java. However, it has come to the game extraordinarily late and, honestly, offers no great distinguishing features that suggest you should use it over any other technology which is equally free.

GLUE is my preferred solution for this purpose – though I have to say that I was only able to get version 2 working easily. Version 3, which is the newest version, did not work for me after about a day's worth of effort. I would recommend, therefore, that you go with version 2 – which is still downloadable – instead.

Assuming that you download version 2, accessing .NET code from Java becomes extraordinarily easy. If you are using JBuilder, there is an IDE plug in that allows you to browse to the URL of a Web Service you want to use, and then click a button… from that point forward you can access your .NET code just as if it were pure Java!

If you aren't using JBuilder, GLUE comes with a command-line tool called *wsdl2java*, which you can run. Simply pass it the URL of your Web Service as a command-line parameter and it will automatically generate some Java files. Include these Java files in your application, a Tomcat web application for example, and you will now be able to write Servlets, Java Server Pages, or anything else you like that can talk seamlessly to .NET!

ASP.NET

Perhaps the most seamless integration of all between .NET and Apache, though, involves the ability of some versions of the Apache web server to actually run ASP.NET. This is made possible via Covalent's commercial "Enterprise Ready Server" – and is only available when Apache is running on a Windows OS where the .NET Framework is available (Windows 2000 or XP, for example).

Basically, there is just a module – called mod_asp.net – which is capable of proxying requests from Apache into the underlying .NET Framework. The responses produced by .NET are then forwarded by Apache to the web client.

Conclusion

This has been just a brief overview of all the different options that are available to you when you have some Apache code that you need to have interoperating with .NET. As I said at the opening to this presentation, I welcome all of your questions and comments to my email – derek@XB.com. Thank you!

URLS

- http://www.microsoft.com/net
- http://www.GotDotNet.com
- http://www.artinsoft.com
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vjsharp/html/vjgrfJavaBinaryConverter.asp
- http://developer.java.sun.com/developer/earlyAccess/j2eecas/download-com-bridge.html
- http://www.jnbridge.com/
- http://jakarta.apache.org/ant/index.html
- http://www.themindelectric.com
- http://www.covalent.net
- http://www.XB.com