

.NET and Java: A Study in Interoperability

By Ted Neward

In the halcyon days of my youth, a major candy manufacturer ran an advertising campaign. It was destined to be a timeless classic, one that sticks with you for the rest of your life: a scene would be some typical meeting place (like a park or a street), one actor would walk from one end of the scene towards the other emerging from the other side. One would be carrying a jar of peanut butter, the other a chocolate bar, and of course, they would bump into each other, the chocolate falling into the peanut butter, and thus a new candy bar was "born". The tagline, of course, was "two great tastes taste great together".

So it goes with Java and .NET. Individually, each is a great platform, but together, a practical necessity in the modern enterprise. With enterprise development market share of 35-40% each, and the Microsoft/Sun settlement agreement of earlier this month, it's fairly obvious that neither of these two platforms is "going away" any time soon. Analysts estimate that by 2005, up to 90% of all IT shops will have both platforms running side by side. It's clear to even the most zealous Java or .NET devotee that working with both platforms is going to become the norm.

This means that developers will be facing an interesting task in the coming years: "make our .NET inventory management system talk to our J2EE customer relationship system", or "our sales staff wants to access our CRM system (written in J2EE) from Outlook", and so on. And while it would be nice to be able to hold up the Web services software stack, point to it and say, "Here's the answer to all your interoperability concerns", the practical and brutal reality of the situation is that it's nowhere near that simple an answer.

Making two platforms interact is at once a simple and difficult problem. Simple, in that it's a fairly closed-requirements solution: if I can work out a few technical details, interaction is achieved. It's also fairly easy to achieve success--if they can talk, you did it, if not, there's still work to go. In fact, once you've worked out low-level issues like byte order, file/data format, and hardware compatibility, basic interaction between any two platforms is pretty straightforward. (As a matter of fact on this basis was the Internet built.)

But the problem of integration still presents hardships, owing to the rich complexity of systems that build on these basic low-level concepts. Merely because I can exchange TCP/IP packets between a Windows machine and a Solaris server does not mean that getting .NET code to exchange data with a J2EE server will be easy--far from it. Numerous hazards lie in wait for the budding integration developer.

Before we get too deep into the interoperability hazards, let's take a moment to revisit Enterprise Application Architecture in general, so as to get our bearings in the discussion to follow.

Historically, we've preferred "n"-tier systems to client/server 2-tier ones, because of the increased scalability intrinsic to "n"-tier systems. An "n"-tier system can scale to much higher user counts, due (among other things) to the shared database connections from a central middle tier to which clients connect. We also prefer the "n"-tier approach because it tends to allow for better separation of responsibilities in code: presentation-layer code goes on the client tier, business logic goes on the middle tier, and data-access code (largely represented by SQL) goes on the resource tiers either on or behind the middle tier machines. 3-tiers, 3 layers, but not always mapped one-to-one.

Drawing a distinction between the tiers and the layers is necessary for good interoperability between the platforms, because interoperability across layers is going to necessitate very different decisions than interoperability within layers. For example, if you have a Windows Forms .NET application that wants to display a Swing applications as a child window, very different decisions are in order than if you want that same Windows Forms applications to talk to a J2EE back end. Web services might suffice for the second requirement; it'll be an unmitigated disaster if you use Web services for the first.

Interoperability can take one of three possible shapes: in-proc, out-of-proc, and resource sharing. Of the three, resource sharing is perhaps the easiest to understand and recognize: using JDBC or ADO.NET, for example, a Java application can write data to a relational database that the .NET program can access via ADO.NET. The database access layers each deal with the necessary details to make the data comprehensible to the appropriate platform, leaving the programmer free to focus on working with the data itself. Unfortunately, a great deal of time is consumed in doing so, and trying to implement a short round-trip request/response cycle using an RDBMS is going to be difficult. Still, as an interoperability mechanism, it's by far the simplest and easiest approach to take, and as a result is the vast majority of the interoperability in production. In essence, we're exchanging data through some well-understood format and easily-accessible medium, in this case the relational database format described by SQL and a central database.

We can extend this raw data exchange in other ways, however, by leveraging the world's most popular interoperable data format, XML, as the format for data exchange and a variety of different data stores as the medium through which the exchange takes place. For example, it's relatively easy to take an XML document and store it to the filesystem from a servlet application for a .NET application to come around and pick up via a filesystem watcher thread. Or, thanks to the growing XML-in/XML-out capabilities of recent database releases, we can use the database as the exchange medium. Or anything else that's handy, for that matter; the key here is that the exchange format is XML data.

This presents its own problems, by the way--XML doesn't deal well with certain aspects of the object-oriented nature of Java and/or C#.NET programs, such as the ability of objects to create cyclical relationships and bidirectional associations. Imagine, for a moment, a Person class like such:

```
public class Person
{
    private Person mother;
    private Person father;
    private ArrayList siblings;
    private Person spouse;
    private ArrayList children;
}
```

While this looks relatively simple to serialize to XML in the simple cases, remember that objects have identity, which means that "Jed's" wife could also be "Jed's" sister in certain parts of the world. Serializing her twice in the XML document breaks object identity and creates further chaos.

SOAP Section 5 of the 1.1 specification sought to correct for this lack, but forgot that not all platforms that can consume XML are object platforms--a good number of them lack any concept of object references whatsoever, in fact. For this reason, SOAP Section 5 encoding is deprecated in later Web services specifications, in favor of XML Schema Definitions (XSD) as a descriptive language for XML data. When building systems that will need to interoperate against any and all possible platforms, always start from schema definitions first, and build object definitions to match against that. Both .NET and Java have libraries and/or specifications to deal with this: the XSD.exe utility and XmlSerializer in .NET, the Java API for XML Binding (JAXB) in Java.

This may all seem redundant--after all, who hasn't heard of XML as the suggested data exchange format? Take careful note, however, that we're talking about data exchange, not the use of XML as a communications stack, which of course brings us to the discussion of interoperability through Web services.

Web Services, as a technology were born of the basic desire to "replicate a call stack in XML" (Don Box, private communication). At the time, it seemed natural enough: take the marshaled parameters from a remote call, and represent them using XML rather than a proprietary binary protocol, thus making it (theoretically) possible for other languages to consume that call without having to write a huge pile of parsing code. Initially, SOAP was available from DevelopMentor and Microsoft in Perl, Java, and COM-based formats.

But as time has progressed, so has the sophistication and scope of Web services. SOAP was rewritten to be a framing and extensibility specification, deferring all question of how to represent data to XSD. Description of Web services endpoints fell to WSDL. Discovery of services was left to UDDI. But

within the last two years, things exploded in complexity; by latest count there are well over 30 specifications from a variety of author entities (Microsoft, IBM and BEA being three of the largest sponsors) covering everything from binary attachments to business process flow. More are coming.

More importantly, when Web services began to sweep the industry as the Next Big Thing, vendor toolkits began to offer extensions that allowed developers to start from the language interface and generate WSDL definitions to make it easy to reuse your existing technology investment by slapping angle brackets around the data traveling across the wire. Unfortunately, any out-of-process remoting API that suggests starting from a language-based interface (whether that be Java or C#) should be taken very skeptically. Consider, for a moment, the following Java interface:

```
public interface Calculator
{
    public BigDecimal add(BigDecimal lhs, BigDecimal rhs);
}
```

what, precisely, should an XML-marshaled BigDecimal instance unmarshal to in .NET?

So often, demos done on the expo show floor clearly prove that the product knows how to talk to the same vendor's product on the other side of the wire, but rarely if ever demonstrates working with another vendor's product. So, for example, an ASMX Web Methods Web service can easily declare itself as returning a Hashtable, for example, but once marshaled and sent across the wire, what format should it resemble in the J2EE space? While Java certainly has its own implementation of Hashtable, there's no love lost between them in implementation details. As a result, it's a fair bet (barring special code to the contrary), the .NET Hashtable will get rendered into a custom data format that has little to no bearing on the .NET Hashtable in widespread use.

For these reasons, just as with data exchange using XSD, when writing WSDL-based services, always start with the "parts in the middle": in this case, the WSDL.

What's worse, few if any of these Web service specifications have a concrete implementation to work with, and fewer still have any sort of "work history" (that is to say, beta-testing and/or field-use) behind them. WS-Routing, for example, exists as an add-on to the Microsoft .NET Framework (Web Services Extensions 1.0), but as of yet no Java Web services software package currently has support for it. Neither Microsoft nor IBM or BEA have any implementation of WS-AtomicTransaction, and so on.

More importantly though, even a perfect Web services picture still leaves the story incomplete. Web services intrinsically imply remote process communication--no WS-* specification currently describes a way for two platforms to coexist in the same process space, for example. (In point of fact, there's really very little that Web services could say; how do you mandate an in-proc API for Python running in the same

process as the CLR, or the JVM?) Interoperability within a given layer (presentation, business logic or data access) sometimes requires the ability to share data in the same process, yet all of the Web services space is focused on simply slinging angle brackets between endpoints across process boundaries. Making things even more interesting, interoperability at the resource tier--the database or the filesystem--is often "just enough" interoperability to make a system work without requiring a major investment.

The most intrusive approach to interoperability is the in-process approach, where a single process hosts both the JVM and the CLR simultaneously. At heart, both managed environments consist of a set of DLLs (JVM.dll in the case of Java, mscorlib.dll in the case of .NET), making it fairly trivial to create a single process hosting both. One simple way to do this in Java, for example, is to write a Java Native Interface-defined method using Microsoft's Managed C++: write the Java native method in the usual manner, generate the C header using javah, then implement and compile the C++ code using the "/clr" switch. Because the JNI DLL is a managed code assembly, the .NET framework will automatically be loaded into the process, as usual, thus bringing both managed environments into the same process. Hosting Java from the CLR is a bit trickier, since it requires the explicit use of the JNI Invocation API, but again it's not rocket science.

There's obviously more to the Web services story than what's been covered here; discussions of particular RPC toolkits and ORBs, messaging frameworks/utilities, SOAP, and more, all even before getting into the Web services discussion. And, what's more, variations and hybrid approaches are quite feasible: a JMS consumer pulling TextMessages out of a Queue that contain a SOAP packet which is delivered via MSMQ by doing JNI methods to Managed C++.... and so on. Just bear in mind when considering your needs and options for your next Java/.NET integration project that multiple options are available to you beyond the traditional "Just set up a WSDL endpoint...."

About Ted Neward

Ted Neward is an independent consultant, author and mentor living in Northern California. He specializes in Java and .NET technologies, particularly in the areas of Java/.NET integration (both in-process and via integration tools like Web services), back-end enterprise software systems, and virtual machine/execution engine plumbing. He is the author or co-author of seven books, including Server-Based Java Programming (Manning Publications, 2000), C# In a Nutshell (O'Reilly, 2001, with Peter Drayton and Ben Albahari), SSCLI Essentials (O'Reilly, with David Stutz and Geoff Shilling), and the forthcoming Effective Enterprise Java (Addison-Wesley). In addition, other technical white papers can be found on the Web sites www.javageeks.com (for Java-related topics) and www.clrgeeks.com (for .NET and CLI related topics), and less-formal technical content can be found at his weblog at <http://www.neward.net/ted/weblog>.

About DevelopMentor

DevelopMentor, a software developer education company, has offered hands-on instructor-led training courses on the latest technologies since 1993. DevelopMentor leads the industry with its unique, immersive Guerrilla training events, public open enrollment courses, custom on-site classes and enterprise-wide skill assessments for experienced software developers across the United States and Europe. DevelopMentor instructors are world-renowned technologists, computer science professionals, authors and courseware experts that teach through demonstration, hands-on labs and real-world application development exercises. Course offerings are part of comprehensive curricula in Microsoft .NET, Java technologies, XML & Web services, Database technologies, Security and other programming-related topics. For more information, please visit www.develop.com or call 1.800.699.1932.