

Best of Breed Enterprise Application Development with SOAP

Edward Garson, Dunstan Thomas Consulting
<http://consulting.dthomas.co.uk>



The nature of software architecture and development has changed significantly and dramatically in the last couple of years. This evolution in systems development has been brought about as the result of a number of factors, most notably the decreasing cost of reliable high-bandwidth internet connectivity and the shortcomings of distributed component-oriented technologies such as Microsoft's DCOM and the OMG's CORBA. Developing distributed component-based systems using these technologies has historically been problematic when one or more participating nodes are located across the internet. Also, the complexity of enterprise application integration (EAI) increases exponentially when these technologies are deployed on heterogeneous platforms (e.g. COM on Windows interfacing with CORBA on Unix). XML came about partially as a response to these problems, and has proven itself to be a near-panacea to the EAI conundrum. Shortly thereafter came SOAP, the equivalent for the integration and development of component-based systems, potentially across heterogeneous platforms.

SOAP is the Simple Object Access Protocol, a.k.a *XML Web Services*, an amalgam of the following:

A *web service* encapsulates an autonomous business process or function, rendered by software accessible via the internet, which may be dynamically discovered, queried and consumed, that fulfils a request from a software agent, to ultimately render all or part of that business process or function to an end user.

Application architectures founded on SOAP interoperability are also known as Service Oriented Architectures (SOA). Each component (or, service) in an SOA is totally independent of any other. They may be aggregated, but are conceptually still indivisible and autonomous, somewhat akin to the transaction paradigm.

Today, stakeholders want it all: rich, graphical clients that integrate tightly with favourite desktop applications, while at the same time enabling mobile (or otherwise) users to access the same business logic via a thin client (e.g. browsers). All this while controlling costs, enabling trusted third parties to access the business logic, and leveraging existing investments in tools, technologies and infrastructure. This is a tall order! Can SOA meet these stringent demands? SOA are in fact able to deliver on a number of what were previously mutually exclusive goals.

The Best of Breed Principle

SOAP enables best of breed application development. The best tools and technologies may be selected for a given environment to create functionally rich presentation-layer clients, while leaving the implementation of server-side tiers totally decoupled from it. Tool and technology choices will invariably be the product of requirements, intellectual investment and in-situ technologies, but the choices when developing SOA-based architectures open up considerably in the light that many tools that historically did not play well together (e.g. Enterprise Java Beans and Visual Basic, to use an extreme example) are now able to do so without difficulty. This is because the presentation-layer (i.e. client-side) communicates with the business tier of an application purely by means of interchanging messages encoded in XML (that's SOAP in a nutshell), so neither the server nor the client care about each other's implementation (which historically has mattered an awful lot).

Best of Breed Choices on the Client Side

It is interesting to note that statistically more than 99% of you will be reading this document on a machine running Microsoft Windows®. Furthermore, this situation is unlikely to change for some time (several years) despite assiduous and fervent development by the open source community (watch this space). Delivering on promises of functionally rich presentation-layer clients (especially for ISV's) therefore means tightly integrating with the defacto client-side operating system of the day. This translates into tool choices including Borland's **Delphi** and Microsoft's **Visual Basic/VB.NET** and the new kid on the block, **C#**. There are very few other realistic choices to consider. Sure, several other tools work well with Windows, but the aforementioned languages have been doing it longer and better than the rest. (What about Visual C++? It's totally convoluted for user interface creation, and that's precisely why VB took off, and the VC++ business logic / VB user interface development paradigm along with it).

So, finding the right tool among these is dependent upon a number of factors including requirements and existing technology investments. All of the following tools have a rich set of features for interacting with SOAP services, all of them are capable of rendering very sophisticated user interfaces, and all play well with Windows, which are our primary concerns in the current context. What differentiates these from one another?

The Case for Borland Delphi

Borland Delphi continues to die a very slow death indeed. The death knoll has been ringing for this tool for a number of years now, yet Delphi continues to surprise us all: this, a testament to just how good this tool really is. Now Borland has committed to adding support for **.NET** in the next version of Delphi, which is interesting as it will be one of very few non-Microsoft development tools to have fully implemented the **.NET** framework. This means that companies that have a significant Delphi investment will be able to enjoy the new extensions being added to an already superlative development tool. On another note, Borland **Delphi** compiles with few modifications on Linux, so if this is a requirement or something that is of interest to you, your choice might already be made.

The Case for Visual Basic / VB.NET

The Visual Basic crowd is somewhat bewildered with the recent changes effected to their language. What was once a powerful environment for the development of user interfaces, it has now become a compiled (well, at least as "compiled" as any interpreted **.NET** assembly can be), fully-fledged object-oriented language in its own right, and many VB developers reacted badly to the significant changes this incurred (witness the backlash this produced). Although these changes were necessary to bring the core of Visual Basic up to scratch for the **.NET** initiative (and hence the change in name from Visual Basic to **VB.NET**), many developers have voted with their feet and moved on to other RAD tools such as **Delphi** and **C#**. Given that the object-oriented paradigm is something that has been bolted onto what was a satisfactory tool for particular aspects of development, **VB.NET** is in fact struggling to find its true identity in this new era of applications development. That being said, **VB** has always been a great tool for building user interfaces, and continues to be a significant player in this arena.

The Case for C#

C# is the newest offering from Redmond on the language front. Although it looks and tastes like Java dressed up in sheep's clothing, **C#** is in fact a totally new offering and is the only language to have been designed with **.NET** in mind from the ground up. Interestingly, the chief architect for **C#** is Anders Hejlsberg, the genius behind **Delphi**, and there is plenty of evidence of his borrowing the best constructs from his previous child. Microsoft also borrowed heavily from the Java community in the creation of their new tool; no surprises there. The result is an elegant language that excels both in the creation of user interfaces while at the same time

being very much at home implementing the business rules. C# will undoubtedly enjoy stellar growth in the coming years as a result.

What about Thin Clients?

The thin client paradigm will never work for enterprise development that mandates a sophisticated user interface (gasp!). This is for one outstanding, overwhelmingly simple and often glossed-over reason: usability. The usability of applications rendered purely by means of HTML (even dynamic) is insufficient for all but the most simple of applications, such as obtaining fiscal quotations, booking airline tickets or managing email. You may argue with this, but the simple proof that thin clients don't work is evidenced by the fact that users do and will continue to install "real" software on their machines. It is categorically impossible to match the functionality of applications like Word and Excel through a browser. Period. For "software as a service" (the .NET mantra) to work in real life, the fact that the user interface is communicating with a remote layer to render functionality has to be totally transparent. This is not the case with browser-based applications. Usability is the Achilles heel of the thin client.

What about Java Clients?

Java clients suffer from the same drawback as that of thin clients, although perhaps less dramatically. The fact that Java user interfaces have to look and behave the same on multitudinous platforms almost totally precludes instantiating native operating system controls (from the perspective of the implementation of the JVM). Java has several operating-system independent user interface toolkits (a.k.a. widgets) including AWT and Swing for exactly this reason. Their independence from the underlying operating system is at the same time their greatest feature and their greatest fault. It is great that these user interfaces are truly "write once, run anywhere" (the Java mantra). It is not great that these user interfaces have awkward shell integration (e.g. drag and drop operations) and play poorly with user's favourite applications (e.g. Office). Finally, they look bad: Java user interfaces are instantly recognizable as such and don't look and feel as slick as natively compiled Windows (or otherwise) executables. That being said, given a requirement to run on a plethora of platforms and less stringent user interface demands, Java is certainly not a fatal choice. The Java community is also working very hard to address these shortcomings: watch this space.

Best of Breed Choices on the Server Side

Now that we have examined some of the best choices for client-side development, it's now time to investigate choices for implementing the server side(!). This would also be a good time to remind ourselves of a fundamental tenet of this paper: that best of breed choices may be made on a per-level or per-tier basis (e.g. presentation layer versus business logic) with impunity due to the ease of interoperability derived from pursuing a service oriented architecture. So, what are the choices in this domain (pun not intended)? There are in fact only two realistic choices: **.NET** or **Enterprise Java** (a.k.a. **J2EE**).

The Case for .NET on the Server Side

The case for .NET on the server is of the same motivation as ever when choosing Microsoft. "Nobody ever got fired for buying Microsoft!" has often been said. Corporations have historically bought into Microsoft for one overwhelming reason: integration. As an example of this, Microsoft was very astute to release the first Office *suite*. Although, when considered individually, none of the products were as good as their stand-alone competitors (witness early versions of Word), the fact that all of these applications played well together was of significant value. Today, the same holds true for corporations that already have a significant investment in Microsoft *on the server side*. Deploying and configuring assemblies on .NET Enterprise Server will be easier to integrate with the rest of the hardware if there is already an investment in Internet Information Server, BizTalk and other Microsoft server-side technologies. Server-side development is also easier because (for example) deploying .NET assemblies over a LAN to a .NET test server is supported by the development environment.

So, choosing to implement SOA using .NET on the server side may simply be a case of "just going with the flow". However, this choice should not be made at the expense of requirements which may outweigh it.

The disadvantage of choosing .NET on the server side is that it precludes running any platform other than Windows to host these components. Many computing professionals are loathe to run Windows servers for reasons that include security and scalability concerns. Historically, and despite the year-old "Trustworthy Computing" initiative, Windows servers and the software they run (e.g. Internet Information Server) are notoriously insecure. Windows servers also scale poorly, and fall over regularly. There are extremely few servers running Windows that have been continually running for one year; but the exact opposite may be said of servers on other platforms. Finally, Windows is not as cheap as some of the competition (read: Linux). These are some of the reasons why Windows has failed to dominate the server side, and so when rejecting this as a platform choice, implementing the server side in .NET goes out the window, too.

The Case for Enterprise Java / J2EE

The battle to dominate the server side is still raging between Microsoft and Unix. Old-school Unix (e.g. SCO) used to be absolutely dominant on the server side, but Microsoft steadily encroached on this market in recent years, primarily due to ease of deployment and low cost of entry (historically, Unix has been very expensive to deploy). The arrival of Linux (new-school Unix) has turned this perception on its head, and is today one of the hottest growth areas of computing. Ignoring this trend means potentially missing out on either an important cost savings or an important source of revenue, depending on your perspective. Gartner and other industry research firms have churned out a plethora of studies promulgating the superior price/performance ratio and lower total cost of ownership when running Linux on enterprise-class servers. IBM doubled Linux-related sales in the last quarter from £75M to £150M+, a testament to stellar growth. Financial institutions are also jumping on the Linux bandwagon in droves, to replace aging and large Unix installations. For tight-fisted IT managers, Linux is manna from heaven. Therefore, rejecting Windows on the server side leads naturally to consider if and where Linux fits into the current discussion.

A very good way to implement web services on the server side is using Enterprise Java (J2EE) components running on Linux. J2EE application servers (software that hosts Java components known as Enterprise Java Beans, or EJB) are massively scalable and also play well with other enterprise tools of note such as Oracle. J2EE also has excellent support for web services development. So, J2EE hosted on Linux combines massive scalability with low total cost of ownership, security and stability to create a combination that is a very attractive choice on the server side indeed.

Conclusion

This paper has examined best of breed application development principles realized by service-oriented architectures. Application tiers may be decoupled from one another to such an extent as to enable cross-platform implementations that leverage the best features of the particular environment and platform that they run on. Now is the time to adopt a SOA-based approach to new applications development. SOA-based architectures are able to combine sophisticated user interfaces with thin-client access potential, while enabling third parties relatively easy access to business logic. With expertise in service oriented architecture development and its underlying technological foundations (XML), Dunstan Thomas is optimally positioned to help customers architect and develop best of breed solutions that meet the stringent demands placed on modern enterprise application development.