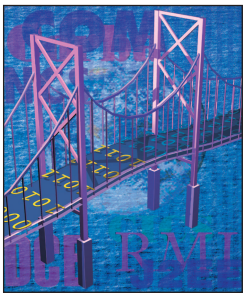
Enterprise Integration with Ruby



Steve Vinoski • *IONA Technologies*

If you've been involved in your share of corporate integration projects, the phrase *enterprise integration* might evoke bad memories. Sadly, some project managers and developers use it to artificially boost their projects' importance, or worse, as an excuse for why their projects are late, over budget, or not delivering required functionality. This phenomenon is somewhat universally understood across the IT industry. Some developers even describe it with the derogatory term *enterprisey*, which is currently so popular that it even has its own Wikipedia entry (http://en.wikipedia.org/wiki/Enterprisey). Fortunately, not all enterprise integration projects are enterprisey; many such projects successfully deliver new capabilities or cost savings.

Continuing the theme in my past two columns, which focused on using dynamic or scripting languages for integration, I take an in-depth look at Maik Schmidt's new book, *Enterprise Integration with Ruby*.[1] His treatment of enterprise integration is eminently practical throughout. For example, he uses the term "enterprise" to mean systems that incorporate multiple technologies and approaches, such as databases, the Lightweight Directory Access Protocol (LDAP), XML, messaging, Remote Procedure Calls (RPC), and distributed objects. If you need to glue such things together, Schmidt offers experiential advice for using the Ruby programming language to do it.

Ruby

Yukihiro "Matz" Matsumoto, of the Network Applied Communication Laboratory, released version 1.0 of Ruby in 1996. He patterned the language after Perl and Python — "ruby" is a play on "perl" — but wanted it to be more object-oriented than either of them. The resulting language's popularity has grown during the past decade, with a recent explosion in interest due to the highly effective Ruby on Rails Web-development framework (www.rubyonrails.org).

Databases

Given databases' ubiquity in the enterprise, the fact that Schmidt starts the book by showing how to use Ruby for database applications isn't surprising. He begins by showing a low level of abstraction involving native database drivers and SQL. Next, he explains how applications can use the Ruby database interface (DBI) to avoid dependencies on specific database drivers. Because of DBI's higher level of abstraction, many enterprisey authors might mindlessly push its use for all cases, instead of native drivers. Schmidt avoids this path, however, and supplies a balanced view of the DBI abstraction's benefits and penalties. He points out that unless your application requires some level of database portability and independence, you might want to avoid the DBI because it carries a performance penalty, and its abstractions could hide important database capabilities. Building enterprise integration applications that actually work often requires making practical trade-offs like these.

The third level of database abstraction explained in the book involves object/relational (O/R) mappings. O/R approaches, like all such mappings, suffer from impedance-mismatch problems because object models and relational models ultimately differ. Despite this issue, Ruby's ActiveRecord module provides a compelling O/R mapping for many applications, and the book includes several examples that show its capabilities. But again, true to form, Schmidt also provides a balanced discussion of ActiveRecord's enterprise readiness, explaining that it might not fit well with legacy database systems that don't follow the conventions on which it relies.

Schmidt also shows how to write Ruby applications that use LDAP to access enterprise informa-

 1089-7801/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society

tion such as personnel and organizational records. Such applications, which bear strong resemblance to database applications, can access LDAP repositories using either the Ruby/LDAP module or the ActiveLDAP module, which is similar in concept and form to the ActiveRecord database module.

XML

Continuing his refreshing theme of practicality, Schmidt explains right off that Ruby supports XML reasonably well but has holes compared to languages such as Java and C#. With expectations properly set, he then dives into how to generate XML documents using raw strings and Ruby modules such as REXML and Builder. Generating XML via raw strings is fraught with problems, as it's easy to miss or incorrectly combine opening and closing tags. Moreover, the intermixing of raw strings and Ruby code for printing them makes the program very hard to read and maintain. Thankfully, REXML and Builder make XML generation much more tractable.

The book also explains how to use REXML to parse XML, using either a tree approach (similar to the Document Object Model [DOM]) or a streaming approach (similar to the Simple API for XML [SAX]). REXML reduces but doesn't eliminate the tedium of writing XML-parsing applications — thus helping explain Schmidt's comments about Ruby's holes in terms of XML handling. However, the book also describes the XmlSimple module, which eliminates some monotony from writing XML-processing code. Instead, the module converts XML documents into data structures comprising hash tables and arrays. This approach simplifies access to XML element and attribute values by using element and attribute names as Ruby hash table keys.

Given that Ruby's XML support isn't groundbreaking when compared to other languages, the book describes several alternatives to XML. The persistent hype surrounding XML has led some developers to believe it's the only way to represent hierarchical data, but that's definitely not the case. For example, more data is probably represented with comma-separated values (CSV, also known as character-separated values) than any other format, and Ruby provides a CSV library to read and write it.

Another format that's popular with dynamic language users is YAML, which stands for "YAML Ain't Markup Language." Ruby also supplies a module for handling this simple, compact, textual format for structured data representation. The book mentions a couple of other alternative formats as well, but I was surprised to see nothing on the JavaScript Object Notation (JSON; www.json.org), given that proponents refer to it as the "fat-free alternative to XML." The popular, easy-to-write data-interchange format is not only available for just about any programming language you can think of, it's also simple enough to parse with a single line of Ruby code.

If you need to glue such things together, Schmidt offers experiential advice for using the Ruby programming language to do it.

Networking and Middleware

Enterprise applications often communicate with each other over networks. Some use simple socket-based communications — an area in which Ruby and other scripting languages excel because they make using sockets so simple. Other enterprise applications rely on middleware based on Corba, Java 2 Enterprise Edition, mainframe technologies, or various messaging approaches.

The book first works through some simple socket examples, which quickly lead to showing how easy it is to write HTTP-based Ruby applications. You'd expect a Ruby HTTP client to be pretty simple, but the author shows that writing applications for the HTTP server side is relatively straightforward as well. The Ruby WEBrick framework handles all the complexities of acting as an HTTP server, leaving the business logic to your application. The WEBrick application model is much like the Java servlet model, in which incoming HTTP requests cause the framework to upcall particular methods on your implementation. In response to an HTTP `GET` request, for example, the WEBrick framework invokes the `do_GET` operation on your Ruby servlet to handle the request. WEBrick also supplies various servlets for common tasks, such as serving files (the FileHandler servlet) and executing CGI scripts (the CGIHandler servlet).

Facilities such as WEBrick make it very easy for Ruby applications to handle HTTP-based interprocess communication approaches, such as XML-RPC and SOAP. If you're forced to communicate with an XML-RPC application or add XML-RPC support to your system, the xmlrpc4r module makes it trivial to do so because it provides a complete implementation of XML-RPC. Similarly, Ruby provides a soap4r library that implements SOAP 1.1. For those who have to integrate with WSDL-based services, soap4r provides a `WSDLDriverFactory` class that takes the name of a WSDL file and generates a client proxy or stub at runtime for the service that the WSDL describes. The book also mentions that soap4r provides a separate tool called wsdl2ruby, which generates Ruby code based on the services spec-

ified in a WSDL document. Many Web services systems provide similar tools that generate Java or C++ code from WSDL. The wsdl2ruby capability could provide developers familiar with these similar tools a good starting point for using Ruby with enterprise applications.

Despite all that the book has to offer, its discussion of Corba is one area with which I have some problems. My experience with Corba dates all the way back to 1991, so I like to think I know a little more about it than the average developer of distributed applications, especially given that I contributed significantly to the Corba specification over the years. First, Schmidt claims that specifications such as Corba were so complex that "only big companies such as Sun, Borland, and IBM had the power to implement [them], and even for them it was sometimes too difficult to do it right." The main issue I have with this statement is that my employer, IONA Technologies, has consistently been the leader in the Corba market since entering it in 1993, and for most of its existence, IONA has employed fewer than 500 people — clearly, implementing Corba isn't limited to big companies.

The book continues its description of Corba: "Consequentially, the situation today is a mess: there are implementations for only a few programming languages, many systems do not interact as they should because of proprietary vendor extensions, and all in all the former 'standards' have been superseded by their young and fresh fellows like XML-RPC anyway."

All three claims in this statement are wrong. First, Corba implementations exist for a wide variety of programming languages, including C, C++, Java, Smalltalk, Lisp, Ada, Cobol, PL/I, Python, and Perl. Second, interoperability between Corba implementations is actually excellent, rather than being as problematic as Schmidt suggests. Finally, I personally don't know of any Corba projects that were replaced with XML-RPC; the two technologies' performance and utility differ so greatly that you can't really meaningfully compare them. Even today, Corba powers many telecommunications and financial applications, and that won't change anytime soon because it's very fast and highly scalable for such applications.

Despite this misinformation about Corba's history and state, the book describes a reasonable approach to integrating Ruby and Corba. Rather than trying to develop a full object-request broker (ORB) in Ruby, the book recommends using the Ruby Java Bridge (RJB) to integrate Corba Java clients with Ruby. RJB lets Ruby applications import Java classes and transparently invoke them through the Java Native Interface (JNI). Such clients can then easily invoke Corba server applications. (The book also mentions another approach: using JRuby, the Ruby implementation for the Java virtual machine (JVM), to embed Ruby in the Corba client.) Schmidt then shows how the Corba client can also be a WEBrick server, effectively providing HTTP access to Corba applications.

REST vs. SOAP

For the past few years, competing camps have debated whether Representational State Transfer (REST) or SOAP provides the more suitable approach for Web services. Roy T. Fielding, now chief scientist at Day Software, first described the REST architectural style in his doctoral dissertation in 2000.[2] REST is widely considered to best describe the World Wide Web's architecture. SOAP, on the other hand, is rooted in RPC-based middleware.

Yet, while the debate has raged on, practitioners have been busy making use of both REST and SOAP. Given the book's pragmatic bent, it's thus no surprise that Schmidt refuses to take sides. Rather than getting involved in what's essentially a religious debate, Schmidt provides the Ruby details you need to deal with both SOAP and REST, given that enterprise developers are likely to see both in practice and rarely have the luxury of choosing sides.

In all, I recommend Maik Schmidt's *Enterprise Integration with Ruby* for enterprise developers. It leaves me no doubt that Ruby can be a powerful ally in enterprise integration wars. The book is well written, provides useful examples (based on a hypothetical flower shop), and explicitly mentions a variety of the typical pain points frequently encountered in enterprise integration projects. If you have a strong Java/C++ middleware integration background but don't know much about scripting, this book will open your eyes to the simplicity and productivity that dynamic languages such as Ruby can offer.

It leaves me no doubt that Ruby can be a powerful ally in enterprise integration wars.

References

1. M. Schmidt, *Enterprise Integration with Ruby*, Pragmatic Bookshelf, 2006.
2. R.T. Fielding, *Architectural Styles and the Design of Network-Based Software Architectures*, doctoral dissertation, Dept. of Computer Science, Univ. of California, Irvine, 2000.

Steve Vinoski is chief engineer for IONA Technologies. He's been involved in middleware for more than 17 years. Vinoski has helped develop middleware standards for the Object Management Group (OMG) and the World Wide Web Consortium (W3C). Contact him at vinoski@ieee.org.