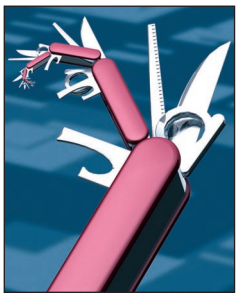

Welcome to "The Functional Web"

By Steve Vinoski • *Verivue*

For me, this magazine issue completes a transition I began two years ago. The previous issue marked the end of the seventh year of the "Toward Integration" column, in which I wrote about issues related to middleware and enterprise integration. One of the points I frequently tried to drive home in that column is that change is inevitable and that it's generally better to embrace it rather than futilely try to prevent it. The transition I'm referring to started when I chose to leave the middleware industry after 16 years. At that point, I initiated the task of refocusing "Toward Integration" toward the Web, which I see as a far more capable, flexible, and cost-effective approach than traditional enterprise middleware for many integration and distribution projects.

With this issue, we bid a final farewell to "Toward Integration" — and with it, the final vestiges of my focus on the world of enterprise middleware — and say hello to "The Functional Web." In this column, I intend to continue writing about Representational State Transfer (REST) and the Web, specifically concentrating on developing production-ready RESTful Web services using functional programming (FP) languages and techniques.

Why Functional Programming?

Despite the fact that FP languages have been around for just about forever in terms of the history of electronic computing — John McCarthy invented Lisp in the late 1950s, for example — they've never shared the popularity or usage levels of their imperative counterparts. For years, the majority of industry-oriented developers considered functional languages to be inefficient and suitable only for academic exercises, and the fact that functional language syntax and idioms differed so widely from what practicing programmers were accustomed to did nothing to help these languages gain popularity. The object-orientation (OO) movement, which started gaining a tangible industry foothold in the 1980s and then, thanks to C++ and Java, boomed tremendously in the 1990s, has resulted in a whole generation of programmers who've grown up with OO programming (OOP) as the primary approach they know and understand. Indeed, OOP is fundamentally the only approach many programmers today really know.

Fortunately, however, FP languages appear to be gaining in popularity for a variety of reasons, most of them centered on different facets of the perpetual themes of performance and efficiency. Perhaps ironically, the resurgence of interest in FP languages owes a lot to Java's popularity. Like many FP languages, Java is based on a virtual machine (VM), and its popularity has driven significant investments in research and development to make VMs more efficient and capable. This has ultimately helped dispel old notions that VMs are bulky and slow. Of course, not all FP languages are VM-based — for example, Objective Caml (OCaml) is among a number of FP languages that can be compiled to either bytecode or native machine code, and it can achieve performance on par with or even exceeding that of C language programs. Nevertheless, FP languages have generally gained much from the significant improvements Java has brought to the VM world.

Another reason for the heightened interest in FP is the move toward multicore architectures. Though not universally agreed upon, some believe that languages based on the mathematical notions of functions, which have no side effects, are better for producing software that can make the most of multicore systems. This belief is based on the notion that code that's free of side effects is easier

continued on p. 102

 1089-7801/09/$25.00

continued from p. 104

to schedule and execute correctly in concurrent threads than code that shares mutable data areas across multiple threads. In Erlang, for example, variables are immutable — once bound to a value, they can't be changed — and there are no global variables. As a result, Erlang can support highly concurrent applications for which developers need not write code to create and manage synchronization and locking among multiple threads. As multicore systems have become more commonplace, interest has risen in FP languages — like Erlang — that offer strong concurrency support.

Still another explanation for FP languages' rising popularity is that developers look to them for the opportunity to get more done with less. Imperative languages such as Java and C++ have come to be known as "high ceremony" languages because of the often mind-numbing amount of syntactic boilerplate and complex object interaction patterns they impose just to get relatively simple applications up and running. Many developers turn to interactive development environments (IDEs) to help them manage this verbosity and complexity, but, in my opinion, this just works around the real problem rather than solves it. By comparison, FP languages are generally far more expressive, so they tend to let programmers state much simpler and briefer solutions.

This last point about programmers getting more done with less, controversial I'm sure, is what attracted me to my current work with FP languages and to Erlang in particular. After years of developing distributed middleware systems in C++ and Java, I figured there simply had to be a better way. Make no mistake, the development teams I worked with were staffed with talented and knowledgeable engineers who were more than capable of developing working, robust middleware frameworks and libraries in Java and C++. Nevertheless, the more experience I gained with helping to build such systems, the more it seemed to me that we invested an excessive amount of time and effort trying to arrange classes and objects into just the right set of abstractions that would enable a quantum leap in our ability to build distributed systems. Despite the fact that our frameworks and libraries "worked" in the sense that they helped their users build distributed applications, I don't believe we ever came up with precisely the right abstractions to actually achieve that quantum leap.

After pondering this problem for years, I finally concluded that our efforts were ultimately most impeded by the programming languages we chose.

After pondering this problem for years, I finally concluded that our efforts were ultimately most impeded by the programming languages we chose. C++ and Java affected how we thought about problems, as well as the shapes of the solutions we came up with, far more deeply than I believe any of us realized. Add to that the verbosity and ceremony of these languages, and the net result is that we wrote, debugged, maintained, and extended a significant amount of code that wasn't directly helping us get to our ultimate goal. We were fundamentally blocked by our inability to change our chosen programming languages into vehicles for application distribution.

Using the wrong languages like this can impose a much larger tax on development efficiency than you might realize. Like the proverbial frog in the pot of water on the stove, eventually boiling to its demise as the water temperature slowly increases because it can't sense the changes until it's too late, developers who primarily use popular imperative languages like Java and C++ can become so accustomed to the boilerplate, verbosity, and ceremony these languages require that they simply don't realize just how inefficient their development efforts really are. Given how defensive such languages' users can often be, perhaps this form of programming language loyalty is a less sinister variant of Stockholm syndrome, where captives counterintuitively develop a sense of devotion and emotional attachment to their captors.

In my experience, the number of lines of code in a system matters a great deal. A system that provides the same capabilities as another but in orders of magnitude fewer lines of code tends to be more straightforward to develop and debug, and vastly easier to maintain and extend. The reason is pretty straightforward: with the smaller system, a much greater chance exists that one developer can keep the whole system in his head. Once a system gets to be so large that no single developer has a hope of understanding all parts of the code, it becomes much harder to preserve system integrity, correctness, extensibility, and overall quality. Judging from my own efforts, FP languages generally allow systems to be stated succinctly and with much less syntactic overhead than imperative languages.

FP: No Stranger to the Web

Using and applying FP languages

and techniques for Web development is nothing new. Over the years, developers have created myriad Web sites using languages such as Perl, Python, and Ruby — each providing features borrowed from the FP world. One important aspect of FP is the notion of higher-order functions in which functions accept other functions as arguments and return functions as results. It's quite common in Ruby, for example, to pass blocks as arguments to other functions. Ruby blocks are essentially anonymous functions that cooperate with the functions that receive them in order to extend and specialize those receivers. Characteristically, this feature is incredibly powerful, given that it allows for very flexible extension and specialization, yet without the verbosity and rigid hierarchy typically imposed by inheritance in OOP.

On the client side, Web developers have over the years made increasing use of JavaScript — a language that's far better and more capable than many give it credit for and one that also incorporates some FP features. JavaScript functions can be passed as arguments and returned as values; anonymous functions and closures are quite commonly used, as are operations such as mapping and folding over lists. These operations are also frequently applied to JavaScript objects, which are essentially tables of name and value pairs.

For cross-browser portability, many Web developers choose to use JavaScript together with a framework or library. My favorite JavaScript library at this time is jQuery (see http://jquery.com), in part because it favors and promotes writing code in an FP style. I intend to cover jQuery in more detail in an upcoming column.

At this time, the language I use for most of my professional software development is Erlang. It was originally designed before the Web came into existence; nevertheless it's garnering a lot of attention these days as a Web service development language. This is because it's excellent for developing distributed server applications that are highly concurrent, very scalable, and amazingly robust — all important Web service properties. Perhaps the most famous Erlang Web server is Yaws (see http://yaws.hyber.org), written starting roughly seven years ago by Claes "Klacke" Wikström, now of Tail-f Systems. Yaws, which is open source and freely available, supports a variety of ways for developers to implement Web services. I'm personally partial to Yaws because I actively contribute to it and help maintain it, so I intend to devote an upcoming column to it, but it's not the only Erlang Web server around. Another solid entry is MochiWeb (see http://code.google.com/p/mochiweb), originally written by Mochi Media's Bob Ippolito and also open source and freely available. It's a favorite of Web service developers who prefer a straightforward, no-frills service framework. Developers use both Yaws and MochiWeb very successfully in production systems.

Developers use both Yaws and MochiWeb very successfully in production systems.

The capabilities and benefits of FP languages like Erlang and Haskell, a pure FP language with strong support for type inferencing as well as considerable control over side effects, have not gone unnoticed in the imperative world. The continuing evolution of the Java VM to a multilanguage platform has resulted in the development of new languages — such as Clojure, a modern Lisp, and Scala, a multiparadigm language that supports FP — that run on that platform. Future columns will discuss these languages, and in particular, the Scala Lift framework (see http://liftweb.net), a Web framework that unites new with old — it lends itself to elegant solutions consisting of just a few lines of code, while also providing access to existing Java libraries.

Of course, RESTful Web service development and deployment have many aspects — not all about writing code. Regardless of what programming language they use, Web developers also have to deal with production concerns such as Web protocol and data format standards, security worries, issues related to integrating with databases and back-end middleware services, and deployment considerations such as scale, uptime, provisioning, upgrades, and logging. Because of the history of FP languages, many developers still view them with suspicion when it comes to such production concerns. Fortunately, FP languages today are generally more production-ready than they're given credit for, so I intend to make sure future "Functional Web" columns address these and other real-world concerns.

The goal of this brand new column is to investigate the application of FP languages and techniques to the world of production-quality RESTful Web service development, but this first column has barely scratched the surface. If there are particular topics or concerns in this area you'd like me to cover here, please don't hesitate to email me.

Steve Vinoski is a member of the technical staff at Verivue in Westford, Mass. He's a senior member of the IEEE and a member of the ACM. You can read Vinoski's blog at http://steve.vinoski.net/blog/ and reach him via vinoski@ieee.org.