REST Eye for the SOA Guy

Steve Vinoski • *IONA Technologies*

You see the glaring headlines all the time in trade magazines, blogs, and mailing-list postings: "REST vs. SOA," "REST vs. SOAP," "REST vs. WS-*." These squabbles can become as strident as religious debates, but I usually find myself taking the middle ground. Several of my 2005 columns, for example, discussed service-oriented architecture (SOA) and dynamic languages, which are often on opposite sides of the fence.

Yet, I'm torn in the Representational State Transfer (REST) and SOA debate — REST is extremely appealing, but my technical background is firmly rooted in the SOA camp. In this column, I try to explain REST from the viewpoint of someone steeped in SOA, with the intention of helping SOA people understand the value the REST camp so rightfully touts.

SOA Basics

Although the main idea behind SOA is valuable, it's hardly Earth-shattering: abstracting your business services and separating them from your applications can yield an overall system that's easier to build, maintain, and extend. This might seem like basic software engineering, but real-world IT systems have countless business rules and assumptions inappropriately buried inside countless monolithic applications. SOA's goal is to avoid such monoliths by separating business rules and policies into distributed services that applications can share as needed.

SOA encourages several critical development practices, but the most important ones are establishing and adhering to service contracts and splitting interface from implementation. A service contract tells its consumers what the service expects as input data when invoked and what form of data it returns. Splitting interface from implementation means that the service's consumers know only the contract and remain blissfully ignorant of its implementation details (such as the programming language in which it's written, the operating system on which it runs, or the service platform within which it executes).

The typical approach to implementing SOA involves more than just services and the applications that use them, though. SOA implementations usually depend on several facilities:

- *service registries*, in which services advertise their locations and capabilities, and where consuming applications go to find those services;
- *service repositories*, in which developers store metadata, such as contract descriptions and policies, for use at both service design and deployment times;
- *service definition languages*, which developers use to define service contracts; and
- *service platforms*, which provide design-time and runtime support for service creation, deployment, and execution.

These descriptions don't mention specific technologies because SOA is applicable to several, despite the fact that many people incorrectly associate it exclusively with SOAP, WSDL, and Web services. Developers have implemented SOA for many years, using technologies such as the Distributed Computing Environment (DCE), Corba, and Java Enterprise Edition (Java EE).

REST Overview

REST is an architectural style that Roy T. Fielding, now chief scientist at Day Software, first defined in his doctoral thesis.[1] Fielding developed REST in 1994, during a time when he also helped develop HTTP 1.0, was the primary architect of HTTP 1.1, and authored the uniform resource identifier (URI) generic syntax. He saw REST as a way to help communicate the basic concepts underlying the Web.

 1089-7801/07/$25.00 © 2007 IEEE

REST specifies several architectural constraints intended to enhance performance, scalability, and resource abstraction within distributed hypermedia systems. One of these is the *uniform interface constraint*, which (as its name implies) means that all resources present the same interface to clients. Another is statelessness, in which servers keep no state on the client's behalf, so all requests must carry the pertinent session-oriented information. Caching is yet another REST architectural constraint that can help performance and scalability by letting clients or intermediaries cache responses that servers mark as cacheable. The fact that the Web works as well as it does is proof of these constraints' effectiveness.

Resources and representations are also key parts of REST — its name is even based on the fact that resources and clients exchange resource representations as part of their interactions. A concrete example is when a Web browser sends an HTTP `GET` request to a Web site for a given resource. The response is typically an HTML representation of the resource's state, but other representations are also possible, such as plaintext or XML. Resources are named with unique identifiers that clients use to interact with them — for the Web, these identifiers are URIs. Because REST targets distributed hypermedia systems, representations also normally contain identifiers for other resources, allowing applications to use them to navigate among related resources. In HTML terms, such identifiers are hyperlinks to related resources.

As doctoral theses go, Fielding's is remarkably readable. If you want to understand the details and motivation behind REST, download or view it at www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. But if you're looking for a quicker REST information fix, or want to know more about how REST has evolved since Fielding published his thesis, the RESTwiki (http://rest.blueoxen.net/) is a good source of relevant information.

Uniform Interfaces and Scalability

SOA proponents regard interfaces and contracts as being critical to service definitions: different services have different interfaces — a normal and desirable characteristic of software systems, whether they're distributed or not. REST proponents, on the other hand, stand by the uniform interface constraint.

One area of agreement between the SOA and REST camps is that loose coupling is generally desirable. It lets different parts of a distributed system evolve at different rates, which both camps agree is absolutely required as system scale increases. Generally, each service in a system has an interface or contract not only for its operations but also for the data exchanged as part of operation invocations. A WSDL definition of a Web service, for example, defines operations in terms of their underlying input and output messages, but it also defines the form of the data that accompanies those messages. For WSDL, data types are normally specified in XML Schema — similarly, Corba services have both interface and data types defined in IDL.

A significant advantage of the uniform interface constraint lies in the area of scalability. For a client to correctly interact with a SOA service, it must understand the specifics of both that service's interface contract and data contract. But for a client to invoke a REST service, it must understand only that service's specific data contract: the interface contract is uniform for all services. I can't overstate this difference's impact on large-scale systems. Imagine, for example, that the Web comprised millions of Web sites and that each defined its own special interface. To use your Web browser to interact with a particular site, you'd likely need to download or write a new browser plug-in that understood that site's interface. Admittedly, I've exaggerated the problem to make its effect clear, but there's no question that the uniform interface constraint can allow for more highly scalable systems. It removes the entire interface contract term from the client–service interaction equation.

Ironically, the fact that SOA prescribes specific interface contracts actually undermines its goal of splitting interface from implementation because specific interfaces tend to reveal more about underlying implementations than do generic interfaces. Moreover, specific interfaces — by definition — constrain their implementations because varying them often requires interface changes. Mark Baker of Coactus Consulting writes about this phenomenon in a couple of online articles (see www.infoq.com/articles/separation-of-concerns and www.coactus.com/blog/2005/11/on-interface-and-implementation-and-reuse/).

Interestingly, some of the architects and developers I know who work on large SOA systems (such as those in telecommunications and financial enterprises) figured out the uniform interface constraint on their own, without ever hearing of REST. Unfortunately, they did it the hard way, by first developing and deploying service-specific interfaces and then watching what broke as their systems increased in scale. From this, they learned two lessons: first, that baking information about a given interface into applications always requires expensive custom coding, and second, that once knowledge of an interface is baked into hundreds or thousands of clients, changing or evolving the interface becomes quite expensive, if not completely impractical. To minimize this effect, they ended up writing generalized interfaces that they could apply to diverse resources, which is essentially what REST's uniform interface constraint prescribes.

Data Variability

Of course, the data variability part of the scalability equation (that different services expect and deliver different data formats) remains within REST,

even if interface variability is eliminated. Although data variability is indeed a factor in both SOA and REST systems, REST has an advantage here as well.

SOA service definition languages often bind data formats together with interface contracts. In Corba IDL, for example, you use the IDL to define your data; with WSDL, developers almost universally use XML Schema for data description, even though WSDL technically supports other approaches as well. This merging of interface and data contracts is based on decisions made decades ago about what service-definition languages should provide for applications, especially in terms of code generation. Today, the SOA camp relies heavily on code generation, tooling, and significant platform and middleware support.

In REST, data formats are necessarily orthogonal to interfaces, given the uniform interface constraint. REST therefore promotes the notion of self-describing messages, in which representation formats are based on agreed-upon standards and are specified within the messages themselves. Different messages can also specify different formats in HTTP `content-type` and `accept` headers — the former indicates the message's data-payload format, whereas the latter specifies as part of a request what data formats the caller is prepared to receive in response.

REST's handling of data formats also helps with scalability. Allowing services to handle multiple data formats means clients and services can use appropriate data types for different types of data, such as images, text, and spreadsheets. Such media types are specific forms of REST's general notion of representation metadata. The fact that such metadata accompanies messages also means that clients can request the data format they'd prefer to receive. Furthermore, the clean separation between distribution infrastructure and representation metadata handling in REST means that developers can build REST-oriented systems using an infrastructure that's much lighter than many SOA platforms.

Resource Naming

Although interface and data contracts obviously vary significantly between REST and SOA, each supports some notion of named resources. REST treats representations as the way for applications to navigate distributed hypermedia systems; similarly, applications normally access SOA services via some sort of distributed network handle. Consequently, the differences between REST and SOA in the area of navigating named resources aren't as great as we might expect. Yet, as far as actually naming resources goes, REST provides much more guidance and consistency than SOA. One of REST's architectural constraints is that each and every resource has a unique identifier. SOA, however, leaves service naming and identification within a particular system entirely up to that system's designer or implementer.

I think the main differences between the REST and SOA camps come down to their respective histories. Both have roots in the distributed objects movement, which decades ago began replacing in-memory object messaging with cross-network object messaging in object-oriented applications. However, they've diverged in their primary focuses. SOA often focuses on application design and construction and is only secondarily concerned with distribution. Look no further than the fact that many modern SOA platforms pride themselves on how easily they let developers turn programming language objects directly into distributed services, regardless of their actual suitability for distribution.

REST's distributed objects heritage is often overlooked, but Fielding noted in his thesis that he originally considered calling REST the "HTTP object model." REST's foremost concern, unlike SOA, has always been distribution: it focuses primarily on ensuring that distributed hypermedia systems can scale and perform well, by explicitly constraining important aspects of their architectures to handle issues related to distribution and by separating critical orthogonal concerns.

Many SOA platforms focus too heavily at the programming-language level and not enough at the distribution level. We've known for years that trying to add distribution after the fact simply doesn't work, but it seems that each batch of SOA developers has to relearn this lesson the hard way. Ironically, the Web's power and ubiquity are making this lesson more difficult because it's simply too easy to improperly tunnel inappropriate protocols through HTTP.

Unfortunately, many SOA developers are far too quick to dismiss REST because it doesn't fit with their favorite tools, because their favorite programming language doesn't directly support it, or because they believe REST is just HTTP. Even if you believe SOA is exactly the right approach for the applications you develop, truly understanding REST can help you build more scalable and better performing distributed systems.

Acknowledgments

Thanks to Mark Baker of Coactus Consulting and Doug Lea of State University of New York, Oswego, for their insightful reviews of the initial draft of this column.

Reference

1. 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 W3C. Contact him at vinoski@ieee.org.