Sponsored by:

JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-10-2008/jw-10-rest-series-1.html

REST for Java developers, Part 1: It's about the information, stupid

A resource-oriented approach to Web services

By Brian Sletten, JavaWorld.com, 10/16/08

Representational State Transfer (REST) is an architectural style for creating, maintaining, retrieving, and deleting resources. REST's information-driven, resource-oriented approach to building Web services can both satisfy your software's users and make your life as a developer easier. This article, the first in a four-part series by REST expert Brian Sletten, introduces the concepts that underlie REST, explains the mechanisms that RESTful applications use, and explores the benefits of REST.

The Web has become the mind-boggling global information ecosystem it is not by accident, but because of specific technology choices made along the way. Roy Fielding, originator of the term REST, documented these choices in his acclaimed Ph.D. thesis. He highlighted the goal of building networked software systems exhibiting the properties of:

- Performance
- Scalability
- Generality
- Simplicity
- Modifiability

Performance, as a property, is a quality of responsiveness. Given the networked nature of these software systems, we always want to avoid paying a latency penalty. Scalability is a property that indicates how many users can simultaneously access a service. Generality allows these systems to solve a wide variety of problems. The more moving parts a software system has and the more complex its interactions, the harder it is to prove that it does what it is supposed to do. We would like our systems to be as simple as possible and extensible in the face of new requirements, new technologies, and new use cases.

Resistance to REST

If asked, most developers would define REST as "Web services with URLs." That is certainly part of the REST vision but by no means all. One reason why they struggle to understand REST concepts is that going "the REST way" often feels like a rejection of the Web service roadmap presented by the World Wide Web Consortium (W3C) and its industry backers. Uncomfortable with this "deviation" from the norm, they have not fully committed to figuring it out.

Another reason why they struggle is that their focus is on the code they write, not on their customers' needs. Most business users and organizations do not care about the software we write, but rather about *information* that software helps them get to.

Technical managers too are uncomfortable deviating from the norm. They prefer the alternative technologies -- SOAP, Web Services Definition Language (WSDL), and various orchestration languages -- because they are entrenched in the industry. Real marketing dollars -- manifest in the tons of books, tools, consulting services, Web pages, and blogs that walk you through these technologies' ever-thickening landscape -- are persuasive. These technologies' specifications are impressively heavy. The notion that REST might be a better way to manage information is perhaps less important to managers than the fact that you can't buy REST or sue someone if it breaks.

Enterprise-system stakeholders would be hard-pressed to deny that they seek these properties in their operational software. The

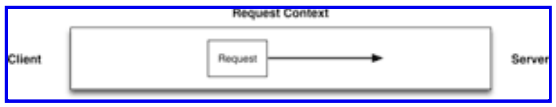
difficulty comes in mapping them to particular technology choices, such as the choice between REST and SOAP.

Invoking behavior vs. managing information

Although there is room for overlap, SOAP is mostly useful for *invoking behavior*, while REST is good for *managing information*. REST can be abused to do SOAPish things, and SOAP can be abused to do RESTful things, but they are fundamentally different technologies. Understanding their differences can lead to appropriate technology choices.

Most applications have traditionally made network requests like the request in Figure 1:

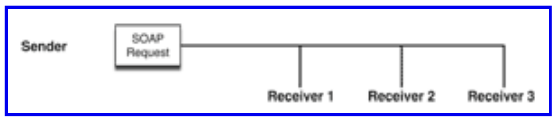
Figure 1. A contextualized request (click to enlarge)



The request is made in a context that can include metadata such as identity, credentials, and preferred response form. The content of the request itself can be kept simple in this model.

SOAP solves the difficult, real problem of making requests that span multiple partners, multiple processing models, and an unpredictable transactional lifetime. In this scenario, a connection cannot stay open for the request's lifetime, so you must decontextualize the request. But once that happens, you need to put the context back into the request. Identity, credentials, security goo, transactional history, and the like accrete around the core request as it moves from receiver to receiver. This model is shown in Figure 2:

Figure 2. A decontextualized request (click to enlarge)



This is a legitimate and important use for SOAP. But it does not necessarily represent a direct approach for managing information. When you want to expose data in language- and platform-independent ways, the SOAP interaction style, with all of its moving parts and complexity, is overkill -- and it lacks some of the important features that the REST style offers.

REST is an architectural style for creating, maintaining, retrieving, and deleting resources. A resource can be anything of potential interest that is serializable in some form -- a file, query, calculation, or concept, for example. This form is called a *representation*. REST, then, refers to the transfer of some bit of information or application state, as a representation, from a server to a client or back again. The REST approach is not better because it is simply different; nor is it different just because it is simple. It is designed to elicit compelling operational properties by applying a series of deliberate constraints. The REST philosophy echoes Albert Einstein's assertion that "the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience." In other words (as Einstein's dictum is often misquoted), "Make things as simple as possible, but no simpler."

Naming resources

REST uses URLs to identify addresses for resources. (This is the first thing most people learn about REST and where some stop, unfortunately.) These names are not intended as arbitrary handles (although URI opacity remains a hotly debated topic), but more as logical paths through *information spaces*. An information space is a vector through the data someone cares about. Each step along the way can mean something in a particular domain. When you try to resolve the name at each level, you expect to get something back -- either a summary or explicit information. As you traverse the path from more generic to more specific, you are actually navigating some shape to the information.

As an example, suppose you are modeling an information system representing a library's inventory. `http://someserver/book` could map to a RESTful engine that would interpret that request as the name of all the books in the library. You may not actually want to retrieve a representation of all of those books (although with pagination, that is clearly an option). Instead, you might return a document representing further classifications of the books or a description of how to invoke the library collection services in different ways. When the request is made from a browser, you might return HTML documentation about how to invoke the service. If you use a programmatic library such as the Restlet client API or Apache HTTP Client, an XML representation of the next level's options might make sense; it could include either simple, popular categories (such as biography, economics, or fiction) or more

formal classification schemes like the Dewey Decimal system.

Information clients would like to navigate the collection however they want. They might prefer to browse by genre, publication date, author, ISBN number, or their own checkout history. REST does not require you to establish a one-to-one mapping between the resources in question and the paths to get there. These are all reasonable ways to find what a library patron might be looking for:

```
http://someserver/book/genre/horror
http://someserver/book/date/2006
http://someserver/book/author/Vonnegut
http://someserver/book/isbn-10/0671799320
http://someserver/book/isbn-13/978-0671799328
http://someserver/book/patron/1234566
```

In this RESTful architecture, it is unlikely that patrons will actually type these URLs (although there is no reason they can't). Rather, they will discover them as they browse categories or view search results. As you return one type of result, you can include hyperlink references to the genre, publication date, and author as well so they simply have to click through to explore. The interaction style of the Web and the interaction style of RESTful services work together nicely.

You can compound these paths when it makes sense, but you should be careful when you try this. `http://someserver/book/patron/1234566/2008` could narrow the list of books returned for a particular patron to those checked out in 2008. You could further refine it by month. However, when you navigate a path, you should not change what *kind* of data is returned. For example, starting out with `/book`, users would expect to receive one or more book results. `http://someserver/book/author/Vonnegut/Kurt` should not return biographical information about the satirist Kurt Vonnegut, but rather a collection of his books maintained by the library. You would want to use a different URL for a biographical information space to return more information about him -- perhaps something like `http://someserver/author/Vonnegut/Kurt`.

The URL in this context:

- Uniquely identifies one or more resources in a resolvable context
- Canonicalizes a path toward an answer that encourages caching

Resolving these URLs is essentially the act of issuing a query. It is easy to cache results when you can give names to them; the RESTful engine can track explicitly what is being asked for and how often. By identifying questions that have already been answered, you avoid pounding the database, which can help reduce back-end costs. Again, with the layered architectures you have been building, you could easily build caching into a persistence layer. If processing (for example, sorting, styling, or filtering) is done on the data, you would have to build separate caching systems to handle each of those states. With REST you can give a logical name to the processed data as well and achieve the same potential for caching there that you did on the raw data. It is known through the logical name, so it can be cached based on the result.

Finally, giving logical names to information allows you to pass *references* to data, rather than the data itself. This not only prevents you from overwhelming middleware with large results; it also prevents orchestrated services from needing to know who has access to what in different application scopes. Any system that tries to resolve the reference can be separately challenged by an authorization and authentication system. Only business systems that have a need to get to sensitive information (such as customer checkout histories) can actually get to it. This helps reduce the burden of role-based access control and increases the fidelity of how you enforce protection of sensitive information. This architecture goes a long way toward reducing the burden of regulatory compliance for use of credit-card, medical-history, or other private data.

URLs also help meet REST's scalability goals, because they address the need for stateless requests. Ultimately, we achieve horizontal scalability by throwing more server hardware at our infrastructure. Any of these servers must be able to handle the request. Any state not maintained in the database or a shared memory system (such as Terracotta) would be lost in the transition from one server to another. That would prevent you from being able to bounce the requests around any piece of hardware. For a request to be stateless, everything needed to satisfy the request must come in as part of the request, including the URL, headers, optional body, and any query parameters. The server that receives this request then has every bit of input necessary to handle it.

REST verbs

Once you have good, logical names that reflect the various shapes of information users care about, you need a way of letting your RESTful service manipulate it. This is where the four REST verbs -- GET, POST, PUT, and DELETE -- come in. Developers coming to this way of thinking from the world of Java APIs and SOAP feel constrained by the availability of only four actions. This reaction is usually a smell that you are thinking about invoking behavior, not manipulating information. GET, POST, PUT and DELETE are suitable for information management. Web services for arbitrary behavior addressed through URLs is not REST. Web services that allow the retrieval and modification of logically named resources that mean something to someone is what REST is about. The HTTP protocol and its verbs are the most common implementation of the REST verbs, but other bindings are not difficult

to imagine.

RESTafarians are quite explicit about keeping names as nouns. It is entirely possible to create a URL like `http://someserver/getemployee&id=12345677` that maps to some back-end query. But if you want to update this resource in an application, you would need to come up with a separate service to manipulate the content. You have forked the relationship between the data and the means to manipulate it. It is suddenly less easy to apply a clear, declarative policy for information-based access control. If instead you use `http://someserver/employee/12345677`, you can issue a GET verb to retrieve the data and update it with PUT, POST, and DELETE.

GET

GET, the most familiar verb, is how your clients ask for the content that they seek. A user either knows a name and types it in directly or receives a link somehow and clicks through. Issuing a GET verb transfers the state of a resource in some representation from a server to a client. The GET verb is called an *idempotent* request, meaning that there are no consequences to issuing it; nothing changes on the server.

When you mix the stateless request style with URLs to identify the resources and the idempotency of the GET request, you achieve a compound key:

```
http://someserver/ + /service + /1234?foo=bar&bat=baz
```

This joint key represents something like a hash key into a computational result set. You can imagine caching all manner of queries and results. Being able to identify these results means you can build infrastructure that does not need to burden the back-end servers. I'll revisit these ideas in a future article in this series.

POST and PUT

It is slightly awkward to talk about POST without also talking about PUT. They seem to do the same thing -- create or update a resource -- but they come from different traditions. POST is from the Usenet community. When people wanted to create a new information resource, say a message about the previous night's "Star Trek" episode, they would create it locally and then POST it to a proxy for the community. No central authority was responsible, so there was no one place to handle the state of the resource. For the same reason, we use POST to create things like orders on e-commerce sites. Until the user transfers an order's state from a browser to the server, the server has no notion of the order's existence or identity. It therefore has no name. As such, we do not know what to call the resource. This is why we have historically POSTed forms to a servlet that processes the request for us. If we have permission to create or update the resource, we will get a response indicating success and perhaps a new address with which to address the resource.

RESTful security

Not all resources can or should support all verbs. Consider what makes sense and who has a business need to manipulate or view the resource. You can control the scope under which your RESTful service responds to verbs based on a request's context. Because your URLs map to named information resources, not arbitrary services, applying a declarative security policy for specific content is straightforward. The details of building this kind of a system are highly implementation-specific, but they easily map into Spring Security, LDAP, single sign-on, and other authentication and authorization systems.

PUT comes to us as part of Tim Berners-Lee's vision of the Web. It is intended as a way to overwrite a named resource. If you create a document that has a specific URL and you want to update it, you can PUT a new version to the URL update the resource.

In general, if you know the name of the resource and wish to overwrite the state, you should probably use a PUT. Otherwise, a POST is a good alternative, particularly if you only want to update a portion of the resource (such as an address or phone number).

DELETE

Thankfully, the public Web doesn't include much support for the DELETE action. Otherwise, factions that differ on various topics might be removing each other's content. On information spaces that you control, however, issuing a DELETE request to a named resource is an important part of the resource's life cycle.

Representation

The final portion of this architectural style is the R in REST: representation. This is the structural form of the resource as it moves back and forth from client to server. In Java and other object-oriented languages, we think about objects, but by giving resources

logical names, you do not have a strict binding between an information consumer and producer and can therefore negotiate the form when you ask for it. This is one of the key advantages of REST over SOAP as far as information management is concerned. You do not want the same physical structure or level of detail back in all situations. XML may be a great format in the business tier, but JavaScript frameworks tend to have difficulty with this data-interchange lingua franca. In a browser-based API, you might like to refer to the same resource but ask for it back as JavaScript Object Notation (JSON). There is no reason, if the server can support the request, for the client to get back anything other than what it needs. With this flexibility in place, however, you can build frameworks that automatically serialize the returned value into Java, C#, or other objects you can use. It is like having the simplicity of Java remote method invocation (RMI) without its Java-only limitations.

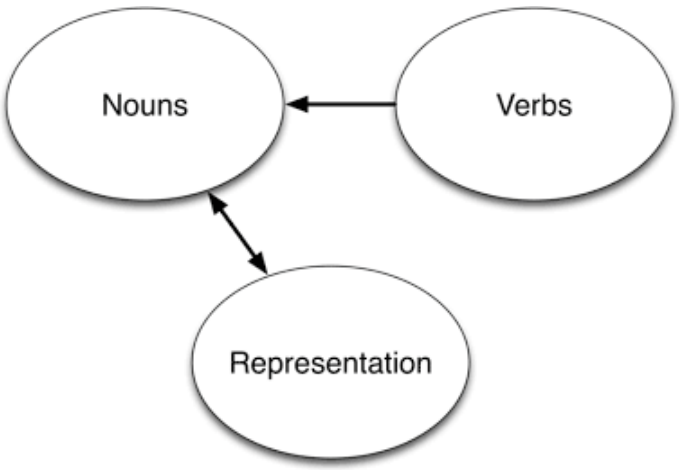
In conclusion

As an architectural style, REST is simple and flexible, and it allows the various communicating pieces to change over time. This flexibility gives you the resilience to embrace the change that inevitably comes in the form of new use cases, new technologies, new requirements, or a new understanding of your domains. Software achieves scalability by talking to logical names that might map to a load balancer that redirects to multiple back-end responders. Clients do not need to know which specific machine they are communicating with. To meet the needs of this kind of an environment, you need an integration approach that separates out the concerns of:

- The things we care about
- How we refer to them
- How we manipulate them
- How we choose to represent them for creation, updates, and retrieval

The simple diagram in Figure 3 shows how REST implements this separation of concerns:

Figure 3. Separation of concerns



WSDL-based SOAP systems do not give you this flexibility. They bind you to a contract that controls what is being asked and of whom, and what will be returned. It is difficult to change any part of that without affecting the clients using the service. REST isn't immune to this problem, but more opportunities to route around it are available. Document/Literal style SOAP has more flexibility than the remote procedure call (RPC) approach, but you lose the ability to name the request and the results the way you can in REST via URLs. This makes it much harder to identify cacheable requests. It is not impossible, and work is being done to improve the situation, but the collection of specifications needed to manage relatively simple requests starts to feel like overengineering.

When you need to invoke behavior in standard, contract-bound ways between disparate partners, SOAP is a good approach. If, on the other hand, you are looking to share information in flexible, scalable, reusable ways, then REST is a great approach. If you would like to build systems that do not require constant migration in the face of back-end flux, then REST's resource-oriented approach gives you that flexibility. The trick is to focus on the information. Who wants it? How do they want it? What do they know and what do they need to know? If you can build software systems that efficiently satisfy these needs, you will add tremendous business value to your customers and give yourself a long-term architectural strategy for success.

In the next article in this series, you'll explore Restlet, an API that makes it easier to build and consume RESTful services in Java.

About the author

Brian Sletten is President of Bosatsu Consulting, Inc. a services company focused on using Web and semantic-oriented technologies

to solve architectural and data-integration problems not handled by conventional tools and techniques. He has a background as a system architect, developer, mentor, and trainer, with experience in the defense, finance, and commercial domains. Brian has a B.S. in computer science from the College of William and Mary and lives in Fairfax, VA.

All contents copyright 1995-2010 Java World, Inc. http://www.javaworld.com