RESTful Web Services Development Checklist

Steve Vinoski • *Verivue*

Sometimes, Representational State Transfer (REST) architectural style proponents describe it as being easy, but this in no way implies that REST is trivial or simplistic, nor does it mean that RESTful systems lack sophistication. REST's relative simplicity comes from the fact that it not only clearly defines its trade-offs and constraints but also distinctly separates concerns, such as resource identification, resource interfaces, and definitions for interchanged data. This delineation makes it relatively easy for developers designing and building RESTful services to consider and track important issues that can profoundly impact system flexibility, scalability, and performance. REST isn't the answer to all distributed computing and integration problems by any stretch of the imagination, but it can yield highly practical solutions to a variety of such problems, not only on the Web but also within the enterprise.

Not everyone agrees that REST is easy, of course. One frequently mentioned issue is a lack of tools — specifically, those that fit within the interactive development environments (IDEs) that many enterprise developers use to help them write and maintain their code. Given that IDEs are helpful only because they automate activities and approaches that developers have already manually proven to be worth automating, this "lack of tooling" argument is somewhat off the mark. With the right language-specific patterns and idioms to follow, existing IDEs work just fine for RESTful service development. For example, it's relatively easy for today's IDEs to incorporate the idioms and patterns from the recently published Java API for RESTful Web Services (JAX-RS) specification (www.jcp.org/en/jsr/detail?id=311) for developing Java-based RESTful Web services. This implies that what's been missing isn't the tooling itself but the specific approaches for how to best develop RESTful Web services in today's popular programming languages; thankfully, as JAX-RS shows, that situation is quickly improving.

Unfortunately, tools can't independently design and implement full systems for us. Whether developers of RESTful HTTP-based services write their code in IDEs or with simple text editors, and regardless of which programming languages they use, they must understand REST and HTTP fundamentals to succeed. This column covers the primary areas that developers must continually consider as they design and build such services. Tools can certainly provide reminders about these areas and help to track progress, but ultimately, developers must understand the underlying technical issues to be able to make suitable design and implementation choices.

Identifiers, Resources, and Applications

RESTful service developers should focus on appropriate resource naming and how servers dispatch requests to resource implementations. As Wikipedia explains in its informative article, "Resource (Web)," (see http://en.wikipedia.org/wiki/Resource_(Web)), a resource is any entity that can be identified or named (that article itself is, in fact, an example of such a resource). Resources are named with URIs. When a request for a given resource arrives, the recipient server decides how the resource's identifier is mapped down to the actual computing entities that implement the resource. Developers new to REST or HTTP often believe that the "path" portion of a URI corresponds to a file system artifact, but this isn't necessarily true, especially when it comes to RESTful services, whose resources tend to be dynamically computed. The flexibility and loose coupling this approach affords is highly beneficial to both client and server, al-

continued on p. 94

continued from p. 96

lowing their respective designs and implementations to evolve in a completely independent manner.

One of REST's most important constraints is *hypermedia as the engine of application state* (HATEOAS), also known as the hypermedia constraint. The relationships between resources and how the server makes those relationships available to applications are at least as important to REST developers as resource naming. As each application uses one or more resources, it maintains its own session state with respect to those. Servers keep resource state, of course, but avoiding the need to keep session or application state on the server side is a big scalability win. Hyperlinks can represent relationships among resources, and as the hypermedia constraint indicates, servers drive applications through viable business logic states via these links.

As important as the hypermedia constraint is, developers don't seem to adequately address it — mainly because it's not how they normally write programs. In typical programming, frameworks and libraries tend to offer numerous method- or function-entry points, such that programmers rarely have to call methods or functions to gain access to further library or framework capabilities. One way to force yourself to think about the hypermedia constraint is to let your service have only one URI as an entry point — a single URI to a single resource that you permit client applications to be aware of a priori. You then have no choice but to consider what URIs a `GET` must return on that single resource to let client applications navigate to other resources; from there, you have to consider what URIs those resources must offer, and so on. Far from a foreign approach, this is precisely the method that most Web sites use to direct interactive browsing from one site page to the next.

Representations and Media Types

RESTful Web service developers must also pay attention to data exchange. The name "Representational State Transfer" means just what it says: RESTful clients and resources transfer resource state representations to each other. The client and server must agree on the formats of such representations, of course, to allow for meaningful exchange.

As I detailed in my March/April 2008 column, HTTP uses MIME media types to identify data formats, which means that developers must consider the nature of their services and decide what MIME types they support. Such types are registered with the Internet Assigned Numbers Authority (IANA; www.iana.org/assignments/media-types), so their definitions are available globally. Developers often turn to general data-definition languages for their services, such as XML or JavaScript Object Notation (JSON; www.json.org).

HTTP supports content negotiation (*conneg*) between clients and services. A client can set the `Accept` header in a request to a list of acceptable MIME types to tell the server what formats it's willing to receive. It can also augment the list with quality (*q*) parameters to indicate preferences. For example, a browser might send an `Accept` header declaring its preference for XHTML, HTML, and image types, in that order, followed by a wildcard indicator with a low *q* parameter to indicate that it will accept anything else as well. Noninteractive programmatic clients, however, tend to prefer a much more limited set of media types — often, just one. When a server returns a response, it sets the `Content-type` header to indicate the type of representation it's returning. To determine the client's preferred content type for a given request, servers must be capable of parsing `Accept` headers using techniques such as those embodied in the open source mimeparse module (http://code.google.com/p/mimeparse/).

Service developers must choose MIME types that work well for their resources and clients. To support the hypermedia constraint, resource representations should contain hyperlinks to related resources wherever it makes sense to do so. Service developers must also ensure that their service implementations return HTTP status code 406, which means "not acceptable," whenever a client requests an unsupported MIME type. Be careful with services that need to support browser access as well as noninteractive client access because at least one browser (Microsoft's Internet Explorer) is notorious for sending `Accept` headers that are essentially useless for determining which MIME type would be best to return. For such cases, you can work around the uninformative `Accept` header by checking the `User-Agent` header to see if the client is the offending browser.

Methods

HTTP provides four basic operations: `GET`, `PUT`, `POST`, and `DELETE`. Developers must consider each method's expected semantics to decide which methods are suitable for each resource. `GET`, `PUT`, and `DELETE`, for example, must be idempotent, and `GET` must be safe for clients to call repeatedly because all it does is return a representation of a resource. The `PUT` method lets a client replace a resource state with a new state, whereas clients use `DELETE` to remove resources. Both obviously have side effects, but both are idempotent because calling them repeatedly has the same effect as calling them once. `POST` can be made to perform virtually any action, but in RESTful systems, it's normally used to create or extend resources, and so it isn't expected to be idempotent or free of side effects.

A common pattern that relies on `POST`, for example, is adding an item to a collection. The client invokes POST on a collection resource, passing along details for the new item in the message body. Assuming those details are OK, the collection resource creates a new item resource and returns its URI in the response's Location header along with status code 201, which means "created."

Status codes are quite important as well. For each method on each resource, developers must choose which HTTP status codes to return, and under what circumstances. The HTTP protocol specification (RFC 2616; www.w3.org/Protocols/rfc2616/rfc2616.html) is clear regarding the meaning of each status code, and clients expect service developers to adhere to and follow those definitions.

However, not all resources support all methods. To determine which ones a given resource can handle, a client can invoke another method called OPTIONS (assuming the resource developer has chosen to support it) to ask the resource directly. The response will normally contain an Allow header that lists methods the resource supports. Should a resource receive a request for a method it doesn't support, it should return status code 405, which means "method not allowed."

Service developers often find creative ways to break expected HTTP method semantics. For example, they might implement GET to have unwanted side effects, such as creating or deleting resources. Of course, developers learn to avoid this when they find their services leaking significant memory as new resources are created on each GET, or they find that their resources are deleted when Web crawlers hit their service URIs. Another frequent blunder is to put a non-idempotent verb, such as "deletePage," into a URI such that accessing it — presumably with a GET — causes the server to perform that action. URIs are names, which are nouns, not verbs. If you think you need to stick a verb for a new method into a URI, chances are quite good that you don't fully understand HTTP's methods, their expected semantics, or how servers and resources can implement them.

Conditional GET

HTTP can be reasonably efficient on a global networking scale because it provides significant support for intermediation and caching. Servers control whether their responses can be cached and, if so, for how long. For further information, refer to the excellent and thorough "Caching Tutorial for Web Authors and Webmasters" by Yahoo's Mark Nottingham (www.mnot.net/cache_docs/). But even for small-scale systems without any caching intermediaries, servers and clients can still include certain data in headers in their responses and requests that can significantly reduce the amount of data they exchange and, in some cases, even eliminate it.

Because conditional GET is relatively straightforward, service developers should always strive to support it. One way to do so is to return the date and time of the most recent change to the resource in the Last-modified header when a client requests a GET of that resource. The next time that client wants to retrieve that same resource, it can take the Last-modified header's value it received last time and send it back to the server in the new request's If-modified-since header. The server then uses this header to see if the resource has changed since the date and time specified by the client; if not, the server returns status code 304, which means "not modified," along with an empty reply body signifying that the client can continue to use the resource representation it originally received. This helps overall efficiency for both the server and client by avoiding sending and receiving the same message bodies repeatedly.

Developers can also support conditional GET through the *entity tags* mechanism. An entity tag uses a resource hash to detect changes rather than relying on date and time, because the latter leaves open a one-second window in which changes can't be detected. A server returns the hash value as a string in the Etag header, and clients can send the hash string back on subsequent requests in the If-none-match header; if the server rehashes the resource and finds that the resulting value matches what the client sent, it returns status code 304 with an empty message body, as it does for the Last-modified case.

Developers must make sure that entity tags' computing cost is much less than the cost of acquiring and returning the whole resource representation. If the resource representation is expensive to compute — requiring multiple database queries, for instance — try to make the entity tag depend on a resource's less expensive subset that's still reliable enough to pick up any changes to it.

In all, REST and HTTP are quite rich. By focusing on the areas I've discussed, RESTful Web service developers can have well-behaved service implementations up and running in short order. For further information, please see Leonard Richardson's and Sam Ruby's excellent RESTful Web Services book.[1]

Acknowledgments

Thanks to Stefan Tilkov of innoQ for his excellent feedback on a draft of this column.

Reference

1. L. Richardson and S. Ruby. *RESTful Web Services*, O'Reilly Media, 2007.

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