IBM

developerWorks Technical topics SOA and web services Technical library

RESTful Web services: The basics

Representational State Transfer (REST) has gained widespread acceptance across the Web as a simpler alternative to SOAP- and Web Services Description Language (WSDL)-based Web services. Key evidence of this shift in interface design is the adoption of REST by mainstream Web 2.0 service providers—including Yahoo, Google, and Facebook—who have deprecated or passed on SOAP and WSDL-based interfaces in favor of an easier-to-use, resource-oriented model to expose their services. In this article, Alex Rodriguez introduces you to the basic principles of REST.

Share:

Alex Rodriguez is a software engineer at IBM focusing on distributed Java technologies and REST Web services. He has been programming Java since JDK 1.1.7B and specializes in designing and developing Java EE-based software.

06 November 2008

Also available in Chinese Japanese Vietnamese

The basics

Develop skills on this topic
This content is part of a progressive knowledge path for advancing your skills. See Build RESTful web services with Java technology

REST defines a set of architectural principles by which you can design Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages. If measured by the number of Web services that use it, REST has emerged in the last few years alone as a predominant Web service design model. In fact, REST has had such a large impact on the Web that it has mostly displaced SOAP- and WSDL-based interface design because it's a considerably simpler style to use.

REST didn't attract this much attention when it was first introduced in 2000 by Roy Fielding at the University of California, Irvine, in his academic dissertation, "Architectural Styles and the Design of Network-based Software Architectures," which analyzes a set of software architecture principles that use the Web as a platform for distributed computing (see Resources for a link to this dissertation). Now, years after its introduction, major frameworks for REST have started to appear and are still being developed because it's slated, for example, to become an integral part of Java™ 6 through JSR-311.

This article suggests that in its purest form today, when it's attracting this much attention, a concrete implementation of a REST Web service follows four basic design principles:

Use HTTP methods explicitly.

Be stateless.

Expose directory structure-like URIs.

Transfer XML, JavaScript Object Notation (JSON), or both.

The following sections expand on these four principles and propose a technical rationale for why they might be important for REST Web service designers.

Use HTTP methods explicitly

One of the key characteristics of a RESTful Web service is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616. HTTP GET, for instance, is defined as a data-producing method that's intended to be used by a client application to retrieve a resource, to fetch data from a Web server, or to execute a query with the expectation that the Web server will look for and respond with a set of matching resources.

REST asks developers to use HTTP methods explicitly and in a way that's consistent with the protocol definition. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:

To create a resource on the server, use POST.

To retrieve a resource, use GET.

To change the state of a resource or to update it, use PUT.

To remove or delete a resource, use DELETE.

An unfortunate design flaw inherent in many Web APIs is in the use of HTTP methods for unintended purposes. The request URI in an HTTP GET request, for example, usually identifies one specific resource. Or the query string in a request URI includes a set of parameters that defines the search criteria used by the server to find a set of matching resources. At least this is how the HTTP/1.1 RFC describes GET. But there are many cases of unattractive Web APIs that use HTTP GET to trigger something transactional on the server—for instance, to add records to a database. In these cases the GET request URI is not used properly or at least not used RESTfully. If the Web API uses GET to invoke remote procedures, it looks like this:

```
GET /adduser?name=Robert HTTP/1.1
```

It's not a very attractive design because the Web method above supports a state-changing operation over HTTP GET. Put another way, the HTTP GET request above has side effects. If successfully processed, the result of the request is to add a new user—in this example, Robert—to the underlying data store. The problem here is mainly semantic. Web servers are designed to respond to HTTP GET requests by retrieving resources that match the path (or the query criteria) in the request URI and return these or a representation in a response, not to add a record to a database. From the standpoint of the intended use of the protocol method then, and from the standpoint of HTTP/1.1-compliant Web servers, using GET in this way is inconsistent.

Beyond the semantics, the other problem with GET is that to trigger the deletion, modification, or addition of a record in a database, or to change server-side state in some way, it invites Web caching tools (crawlers) and search engines to make server-side changes unintentionally simply by crawling a link. A simple way to overcome this common problem is to move the parameter names and values on the request URI into XML tags. The resulting tags, an XML representation of the entity to create, may be sent in the body of an HTTP POST whose request URI is the intended parent of the entity (see Listings 1 and 2).

Listing 1. Before

```
GET /adduser?name=Robert HTTP/1.1
```

Listing 2. After

```
POST /users HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>
  <name>Robert</name>
</user>
```

The method above is exemplary of a RESTful request: proper use of HTTP POST and inclusion of the payload in the body of the request. On the receiving end, the request may be processed by adding the resource contained in the body as a subordinate of the resource identified in the request URI; in this case the new resource should be added as a child of `/users`. This containment relationship between the new entity and its parent, as specified in the POST request, is analogous to the way a file is subordinate to its parent directory. The client sets up the relationship between the entity and its parent and defines the new entity's URI in the POST request.

A client application may then get a representation of the resource using the new URI, noting that at least logically the resource is located under `/users`, as shown in Listing 3.

Listing 3. HTTP GET request

```
GET /users/Robert HTTP/1.1
Host: myserver
Accept: application/xml
```

Using GET in this way is explicit because GET is for data retrieval only. GET is an operation that should be free of side effects, a property also known as *idempotence*.

A similar refactoring of a Web method also needs to be applied in cases where an update operation is supported over HTTP GET, as shown in Listing 4.

Listing 4. Update over HTTP GET

```
GET /updateuser?name=Robert&newname=Bob HTTP/1.1
```

This changes the `name` attribute (or property) of the resource. While the query string can be used for such an operation, and Listing 4 is a simple one, this query-string-as-method-signature pattern tends to break down when used for more complex operations. Because your goal is to make explicit use of HTTP methods, a more RESTful approach is to send an HTTP PUT request to update the resource, instead of HTTP GET, for the same reasons stated above (see Listing 5).

Listing 5. HTTP PUT request

```
PUT /users/Robert HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>
  <name>Bob</name>
</user>
```

Using PUT to replace the original resource provides a much cleaner interface that's consistent with REST's principles and with the definition of HTTP methods. The PUT request in Listing 5 is explicit in the sense that it points at the resource to be updated by identifying it in the request URI and in the sense that it transfers a new representation of the resource from client to server in the body of a PUT request instead of transferring the resource attributes as a loose set of parameter names and values on the request URI. Listing 5 also has the effect of renaming the resource from `Robert` to `Bob`, and in doing so changes its URI to `/users/Bob`. In a REST Web service, subsequent requests for the resource using the old URI would generate a standard 404 Not Found error.

As a general design principle, it helps to follow REST guidelines for using HTTP methods explicitly by

using nouns in URIs instead of verbs. In a RESTful Web service, the verbs—POST, GET, PUT, and DELETE—are already defined by the protocol. And ideally, to keep the interface generalized and to allow clients to be explicit about the operations they invoke, the Web service should not define more verbs or remote procedures, such as `/adduser` or `/updateuser`. This general design principle also applies to the body of an HTTP request, which is intended to be used to transfer resource state, not to carry the name of a remote method or remote procedure to be invoked.

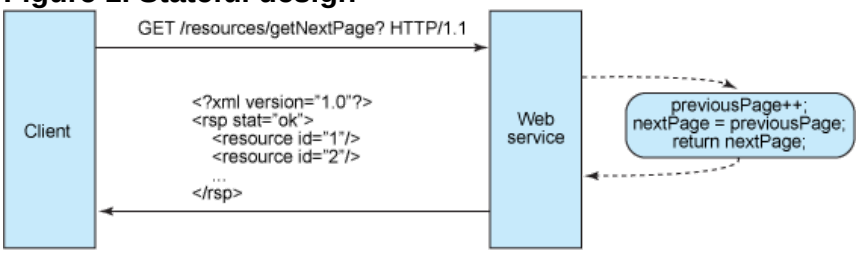
Be stateless

REST Web services need to scale to meet increasingly high performance demands. Clusters of servers with load-balancing and failover capabilities, proxies, and gateways are typically arranged in a way that forms a service topology, which allows requests to be forwarded from one server to the other as needed to decrease the overall response time of a Web service call. Using intermediary servers to improve scale requires REST Web service clients to send complete, independent requests; that is, to send requests that include all data needed to be fulfilled so that the components in the intermediary servers may forward, route, and load-balance without any state being held locally in between requests.

A complete, independent request doesn't require the server, while processing the request, to retrieve any kind of application context or state. A REST Web service application (or client) includes within the HTTP headers and body of a request all of the parameters, context, and data needed by the server-side component to generate a response. Statelessness in this sense improves Web service performance and simplifies the design and implementation of server-side components because the absence of state on the server removes the need to synchronize session data with an external application.

Figure 1 illustrates a stateful service from which an application may request the next page in a multipage result set, assuming that the service keeps track of where the application leaves off while navigating the set. In this stateful design, the service increments and stores a `previousPage` variable somewhere to be able to respond to requests for next.

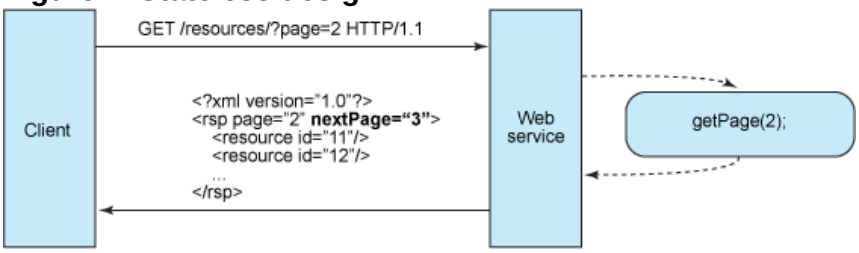
Figure 1. Stateful design



Stateful services like this get complicated. In a Java Platform, Enterprise Edition (Java EE) environment stateful services require a lot of up-front consideration to efficiently store and enable the synchronization of session data across a cluster of Java EE containers. In this type of environment, there's a problem familiar to servlet/JavaServer Pages (JSP) and Enterprise JavaBeans (EJB) developers who often struggle to find the root causes of `java.io.NotSerializableException` during session replication. Whether it's thrown by the servlet container during `HttpSession` replication or thrown by the EJB container during stateful EJB replication, it's a problem that can cost developers days in trying to pinpoint the one object that doesn't implement `Serializable` in a sometimes complex graph of objects that constitute the server's state. In addition, session synchronization adds overhead, which impacts server performance.

Stateless server-side components, on the other hand, are less complicated to design, write, and distribute across load-balanced servers. A stateless service not only performs better, it shifts most of the responsibility of maintaining state to the client application. In a RESTful Web service, the server is responsible for generating responses and for providing an interface that enables the client to maintain application state on its own. For example, in the request for a multipage result set, the client should include the actual page number to retrieve instead of simply asking for *next* (see Figure 2).

Figure 2. Stateless design



A stateless Web service generates a response that links to the next page number in the set and lets the client do what it needs to in order to keep this value around. This aspect of RESTful Web service design can be broken down into two sets of responsibilities as a high-level separation that clarifies just how a stateless service can be maintained:

Server

Generates responses that include links to other resources to allow applications to navigate between related resources. This type of response embeds links. Similarly, if the request is for a parent or container resource, then a typical RESTful response might also include links to the parent's children or subordinate resources so that these remain connected.

Generates responses that indicate whether they are cacheable or not to improve performance by reducing the number of requests for duplicate resources and by eliminating some requests entirely. The server does this by including a Cache-Control and Last-Modified (a date value) HTTP response header.

Client application

Uses the Cache-Control response header to determine whether to cache the resource (make a local copy of it) or not. The client also reads the Last-Modified response header and sends back the date value in an If-Modified-Since header to ask the server if the resource has changed. This is called Conditional GET, and the two headers go hand in hand in that the server's response is a standard 304 code (Not Modified) and omits the actual resource requested if it has not changed since that time. A 304 HTTP response code means the client can safely use a cached, local copy of the resource representation as the most up-to-date, in effect bypassing subsequent GET requests until the resource changes.

Sends complete requests that can be serviced independently of other requests. This requires the client to make full use of HTTP headers as specified by the Web service interface and to send complete representations of resources in the request body. The client sends requests that make very few assumptions about prior requests, the existence of a session on the server, the server's ability to add context to a request, or about application state that is kept in between requests.

This collaboration between client application and service is essential to being stateless in a RESTful Web

service. It improves performance by saving bandwidth and minimizing server-side application state.

Expose directory structure-like URIs

From the standpoint of client applications addressing resources, the URIs determine how intuitive the REST Web service is going to be and whether the service is going to be used in ways that the designers can anticipate. A third RESTful Web service characteristic is all about the URIs.

REST Web service URIs should be intuitive to the point where they are easy to guess. Think of a URI as a kind of self-documenting interface that requires little, if any, explanation or reference for a developer to understand what it points to and to derive related resources. To this end, the structure of a URI should be straightforward, predictable, and easily understood.

One way to achieve this level of usability is to define directory structure-like URIs. This type of URI is hierarchical, rooted at a single path, and branching from it are subpaths that expose the service's main areas. According to this definition, a URI is not merely a slash-delimited string, but rather a tree with subordinate and superordinate branches connected at nodes. For example, in a discussion threading service that gathers topics ranging from Java to paper, you might define a structured set of URIs like this:

```
http://www.myservice.org/discussion/topics/{topic}
```

The root, `/discussion`, has a `/topics` node beneath it. Underneath that there are a series of topic names, such as gossip, technology, and so on, each of which points to a discussion thread. Within this structure, it's easy to pull up discussion threads just by typing something after /topics/.

In some cases, the path to a resource lends itself especially well to a directory-like structure. Take resources organized by date, for instance, which are a very good match for using a hierarchical syntax.

This example is intuitive because it is based on rules:

```
http://www.myservice.org/discussion/2008/12/10/{topic}
```

The first path fragment is a four-digit year, the second path fragment is a two-digit day, and the third fragment is a two-digit month. It may seem a little silly to explain it that way, but this is the level of simplicity we're after. Humans and machines can easily generate structured URIs like this because they are based on rules. Filling in the path parts in the slots of a syntax makes them good because there is a definite pattern from which to compose them:

```
http://www.myservice.org/discussion/{year}/{day}/{month}/{topic}
```

Some additional guidelines to make note of while thinking about URI structure for a RESTful Web service are:

- Hide the server-side scripting technology file extensions (.jsp, .php, .asp), if any, so you can port to something else without changing the URIs.
- Keep everything lowercase.
- Substitute spaces with hyphens or underscores (one or the other).

Avoid query strings as much as you can.

Instead of using the 404 Not Found code if the request URI is for a partial path, always provide a default page or resource as a response.

URIs should also be static so that when the resource changes or the implementation of the service changes, the link stays the same. This allows bookmarking. It's also important that the relationship between resources that's encoded in the URIs remains independent of the way the relationships are represented where they are stored.

Transfer XML, JSON, or both

A resource representation typically reflects the current state of a resource, and its attributes, at the time a client application requests it. Resource representations in this sense are mere snapshots in time. This could be a thing as simple as a representation of a record in a database that consists of a mapping between column names and XML tags, where the element values in the XML contain the row values. Or, if the system has a data model, then according to this definition a resource representation is a snapshot of the attributes of one of the things in your system's data model. These are the things you want your REST Web service to serve up.

The last set of constraints that goes into a RESTful Web service design has to do with the format of the data that the application and service exchange in the request/response payload or in the HTTP body. This is where it really pays to keep things simple, human-readable, and connected.

The objects in your data model are usually related in some way, and the relationships between data model objects (resources) should be reflected in the way they are represented for transfer to a client application. In the discussion threading service, an example of connected resource representations might include a root discussion topic and its attributes, and embed links to the responses given to that topic.

Listing 6. XML representation of a discussion thread

```
<?xml version="1.0"?>
<discussion date="{date}" topic="{topic}">
  <comment>{comment}</comment>
  <replies>
    <reply from="joe@mail.com" href="/discussion/topics/{topic}/joe"/>
    <reply from="bob@mail.com" href="/discussion/topics/{topic}/bob"/>
  </replies>
</discussion>
```

And last, to give client applications the ability to request a specific content type that's best suited for them, construct your service so that it makes use of the built-in HTTP Accept header, where the value of the header is a MIME type. Some common MIME types used by RESTful services are shown in Table 1.

Table 1. Common MIME types used by RESTful services

MIME-Type	Content-Type
JSON	application/json
XML	application/xml
XHTML	application/xhtml+xml

This allows the service to be used by a variety of clients written in different languages running on different platforms and devices. Using MIME types and the HTTP Accept header is a mechanism known as

content negotiation, which lets clients choose which data format is right for them and minimizes data coupling between the service and the applications that use it.

Conclusion

REST is not always the right choice. It has caught on as a way to design Web services with less dependence on proprietary middleware (for example, an application server) than the SOAP- and WSDL-based kind. And in a sense, REST is a return to the Web the way it was before the age of the big application server, through its emphasis on the early Internet standards, URI and HTTP. As you've examined in the so-called principles of RESTful interface design, XML over HTTP is a powerful interface that allows internal applications, such as Asynchronous JavaScript + XML (Ajax)-based custom user interfaces, to easily connect, address, and consume resources. In fact, the great fit between Ajax and REST has increased the amount of attention REST is getting these days.

Exposing a system's resources through a RESTful API is a flexible way to provide different kinds of applications with data formatted in a standard way. It helps to meet integration requirements that are critical to building systems where data can be easily combined (mashups) and to extend or build on a set of base, RESTful services into something much bigger. This article touches on just the basics here but hopefully in a way that has enticed you to continue exploring the subject.

Resources

Learn

Read chapter 5 of Roy Fielding's dissertation, "Architectural Styles and the Design of Network-based Software Architectures."

Get more information about RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1

Read the book *RESTful Web Services*.

The SOA and Web services zone on IBM developerWorks hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on how to develop Web services applications.

The IBM SOA Web site offers an overview of SOA and how IBM can help you get there.

Check out a quick Web services on demand demo.

Get products and technologies

Download JSR 311: JAX-RS: The Java API for RESTful Web Services.

Get Jersey (GlassFish).

Download Restlet, the REST framework for Java.

Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

Get involved in the developerWorks community by participating in developerWorks blogs.

Dig deeper into SOA and web services on developerWorks

- Overview
- New to SOA and web services
- Technical library (tutorials and more)
- Downloads and products
- Open source projects
- Standards
- Events