

A multi-tier architecture for building RESTful Web services

Skill Level: Intermediate

[Bruce Sun \(bsun@ucar.edu\)](mailto:bsun@ucar.edu)

Java Architect

National Center for Atmospheric Research

09 Jun 2009

RESTful Web services have emerged as a promising alternative to SOAP-based services due to their simplicity, lightweight nature, and the ability to transmit data directly over HTTP. In this article, get an overview of the concept of REST and RESTful Web services, then compare them to RPC-style/SOAP-based Web services. You'll also learn about Java™ frameworks for building RESTful Web services as well as a shared multi-tier architecture for building both RESTful Web services and dynamic Web applications.

Introduction

It is imperative that modern Web applications provide desktop-like rich browser interfaces using Asynchronous JavaScript and XML (Ajax) or the Google Web Toolkit (GWT) as well as RESTful Web services to external client applications. This article proposes a Resource Request Handler (RRH) for Ajax/GWT and calls from external client applications, and a Browser Request Handler (BRH) for processing requests from the browser and generating output to display in the browser. Both handlers share a common Business Logic Layer, which in turn interacts with the Data Access Layer. The extraction of RRH and BRH simplifies the design and helps facilitate code reuse, leading to a flexible and extensible architecture.

What is REST?

REST (REpresentation State Transfer) describes an architectural style of networked systems such as Web applications. It was first introduced in 2000 in a Ph.D

dissertation by Roy Fielding, one of the principal authors of the HTTP specification. REST refers to a collection of architecture constraints and principles. An application or design, if it meets those constraints and principles, is RESTful.

One of the most important REST principles for Web applications is that the interaction between the client and server is stateless between requests. Each request from the client to the server must contain all of the information necessary to understand the request. The client wouldn't notice if the server were to be restarted at any point between the requests. Additionally, stateless requests are free to be answered by any available server, which is appropriate for an environment such as cloud computing. The client can cache the data to improve performance.

On the server side, the application state and functionality are divided into resources. A resource is an item of interest, a conceptual identity that is exposed to the clients. Example resources include application objects, database records, algorithms, and so on. Every resource is uniquely addressable using a URI (Universal Resource Identifier). All resources share a uniform interface for the transfer of state between client and server. Standard HTTP methods such as GET, PUT, POST, and DELETE are used. Hypermedia is the engine of the application state, and resource representations are interconnected by hyperlinks.

Another important REST principle is the *layered system*, which means a component cannot see beyond the immediate layer with which it is interacting. By restricting knowledge of the system to a single layer, a boundary is placed on the overall system complexity, promoting substrate independence.

REST architectural constraints, when applied as a whole, generate an application that scales well to a large number of clients. It also reduces interaction latency between clients and servers. The uniform interface simplifies the overall system architecture and improves the visibility of the interactions between subsystems. REST simplifies implementation for both the client and server.

RESTful Web services versus RPC-style Web services

Until recently, SOAP-based Web services built with RPC-style architecture have been the most popular approach to implementing a Service-oriented architecture (SOA). RPC-style Web service clients send an envelope full of data, including method and argument information, to the server over HTTP. The server unwinds the envelope and executes the named methods with the passed arguments. The results of the method are packed into an envelope and sent back to the client as a response. The client receives the response and unwinds the envelope. Every object has its own unique methods and the RPC-style Web service exposes only one URI, which represents the single end point. It ignores most features of HTTP and supports only the POST method.

The RESTful approach to Web services is emerging as a popular alternative due to its lightweight nature and the ability to transmit data directly over HTTP. Clients are implemented using a wide variety of languages such as Java programs, Perl, Ruby, Python, PHP, and Javascript (including Ajax). RESTful Web services are generally accessed by an automated client or an application acting on behalf of the user. However, their simplicity makes it possible for humans to interact with them directly, constructing a GET URL with their Web browser and reading the content that is returned.

In a REST-style Web service, every resource has an address. Resources themselves are the targets for method calls, and the list of methods is the same for all resources. The methods are standard including HTTP `GET`, `POST`, `PUT`, `DELETE`, and possibly `HEADER` and `OPTIONS`.

In an RPC-style architecture, the focus is on methods, while in REST-style architecture, the focus is on resources—pieces of information to be retrieved as a representation and manipulated using standard methods. Resource representations are interconnected by the hyperlinks within the representation.

Leonard Richardson & Sam Ruby introduce a term `REST-RPC` hybrid architecture in their book *RESTful Web Services*. Instead of using envelopes to wrap the method, arguments, and data, a REST-RPC hybrid Web service transmits the data directly over HTTP, which is similar to the REST-style. But it doesn't use standard HTTP methods for operations on the resources. It stores the method information in the URI portion of the HTTP request. Several well-known Web services such as Yahoo's Flickr API and the del.icio.us API use this hybrid architecture.

Java frameworks for RESTful Web services

Two Java frameworks have emerged to help with building RESTful Web services. Restlet (see [Resources](#)) by Jerome Louvel and Dave Pawson is lightweight. It implements concepts such as resources, representation, connector, and media type for any kind of RESTful system, including Web services. In the Restlet framework, both the client and server are components. Components communicate with each other through connectors. The most important classes in the framework are the abstract class `Uniform` and its concrete subclass `Restlet`, the subclasses of which are specialized classes such as `Application`, `Filter`, `Finder`, `Router`, and `Route`. Those subclasses work together to handle authentication, filtering, security, data transformation, and routing the incoming requests to the respective resources. The `Resource` class generates the presentation for the client.

JSR-311 (see [Resources](#)) is a specification from Sun Microsystems that defines a set of Java APIs for the development of RESTful Web services. Jersey (see [Resources](#)) is the reference implementation for JSR-311.

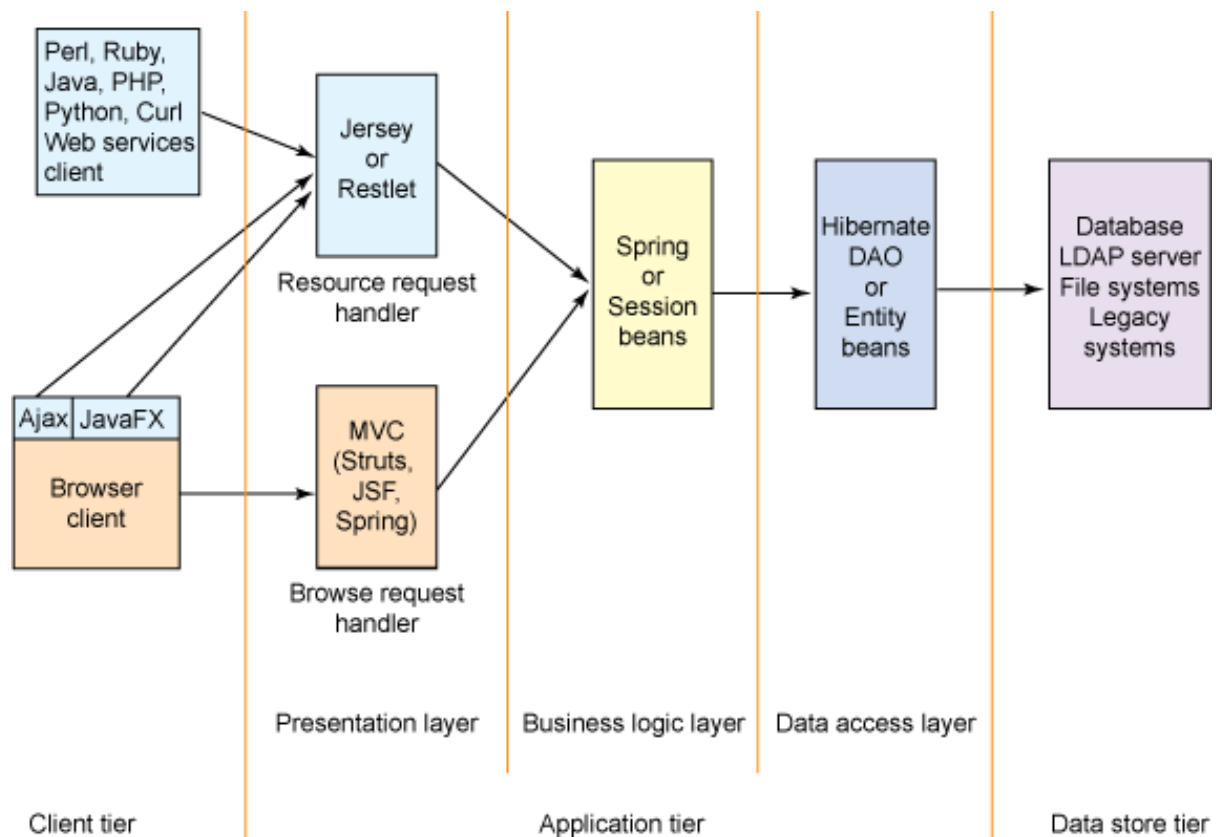
JSR-311 provides a set of annotations with associated classes and interfaces that can be used to expose Java objects as Web resources. The specification assumes HTTP is the underlying network protocol. It provides clear mappings between the URI and corresponding resource classes, and mappings between HTTP methods with the methods in Java objects, by using annotations. The API supports a wide range of HTTP entity content types including HTML, XML, JSON, GIF, JPG, and so on. It will also provide the needed plug-in ability to allow additional types to be added by an application in a standard manner.

A multi-tier architecture for building RESTful Web services

RESTful Web services and dynamic Web applications are similar in many ways. Sometimes they provide the same or very similar data and functions, albeit to different kinds of clients. For example, an online e-commerce catalog Web site provides a browser interface for users to search, view, and order products. It would be helpful if it also provided Web services so that companies, retailers, or even individuals can order the products automatically. Web services can benefit from the separation of concerns inherent in a multi-tier architecture in much the same way as most dynamic Web applications. The business logic and data can be shared by both automated and GUI clients. The only differences are the nature of the client and the presentation layer of the middle tier. In addition, separating business logic from data access enables database-independence and provides for plug-ability to various types of data stores.

Figure 1 shows automated clients, which include Java and scripts of various languages such as Python, Perl, Ruby, PHP, or command-line tools such as curl. Ajax, Flash, JavaFX, GWT, blogs, and wikis that run inside the browser and act as RESTful Web service consumers also belong to this group because they act in an automated fashion on behalf of the user. Automated Web service clients send HTTP requests to the Resource Request Handler in the Web tier. The stateless requests from the client contain the method information in the header, namely `POST`, `GET`, `PUT`, and `DELETE`, which will be mapped to corresponding operations of the resources in the Resource Request Handler. Each request contains all the necessary information including credentials for the Resource Request Handler to process the request.

Figure 1. Diagram of a multi-tiered Web application environment



After receiving a request from a Web service client, the Resource Request Handler requests the service from the business logic layer. The Resource Request Handler identifies all the conceptual entities that the system exposes as resources and assigns a unique URI to each of them. The conceptual identities, however, don't exist in this layer. Instead, they exist in the business logic layer. The Resource Request Handler can be implemented using Jersey or another framework such as Restlet and should be lightweight, mainly delegating the heavy duty work to the business tier.

Ajax and RESTful Web services naturally fit with each other. They both leverage widely available Web technologies and standards such as HTML, JavaScript, browser objects, XML/JSON, and HTTP. There is absolutely no need to buy, install, or configure any other major component to enable effective interaction between Ajax front ends and RESTful Web services. RESTful Web services provide Ajax with a very simple API to deal with the interactions with resources on the server.

The Web browser client in Figure 1 acts as a GUI front end providing display functions using HTML generated by the Browser Request Handler in the presentation layer. The Browser Requester Handler can be implemented using MVC models (JSF, Struts, or Spring are Java examples). It accepts the request from the browser, requests the service from the business logic layer, generates the presentation, and responds to the browser. The presentation is intended to be

displayed to the user by the browser. The presentation contains not just the content, but also the attributes for display such as HTML and CSS.

Business rules are centralized into the business logic layer that serves as an intermediary for data exchange between the presentation layer and the data access layer. Data is provided to the presentation layer in the form of domain objects or value objects. Decoupling the Browser Request Handler and Resource Request Handler from the business logic layer helps facilitate code reuse, and leads to a flexible and extensible architecture. In addition, as new REST and MVC frameworks become available in the future, it is easier to implement them without rewriting the business logic layer.

The data access layer provides the interface with the data store tier and can be implemented using the DAO design pattern or object-relational mapping solutions such as Hibernate, OJB or iBATIS. As an alternative, the components in the business layer and data access layer can be implemented as EJB components with support from an EJB container that facilitates the component life cycle and manages the persistence, transactions and resource allocations. However, this does require a Java EE-compliant application server such as JBoss and would not work with Tomcat. The power of this layer is in separating data access code from the business logic for disparate data store technologies. The data access layer can also act as an integration point to link with other systems, including being a client of other Web services.

The data store tier includes database systems, LDAP servers, file systems, and enterprise information systems including legacy systems, transaction processing systems, and enterprise resource planning systems. Using this architecture, you can begin to see the power of a RESTful Web service, which has the flexibility to be a unified API to any enterprise data store, thus exposing stove-piped data to user-centric Web applications and automated batch reporting scripts.

Conclusion

REST describes an architectural style of networked systems such as Web applications. REST constraints, when applied as a whole, generate a simple, scalable, efficient, secure, reliable, and extensible architecture. RESTful Web services have emerged as a promising alternative to SOAP-based services due to their simplicity, lightweight nature, and the ability to transmit data directly over HTTP. A multi-tier architecture for both Web services and dynamic Web applications leads to reusability, simplicity, extensibility, and clear separation of component responsibilities. Ajax and RESTful Web services fit naturally with each other. Developers can easily create rich interfaces by using Ajax and RESTful Web services together.

This article is a precursor to a tutorial being written by the author to demonstrate

how to build RESTful Web services and dynamic Web applications with the multi-tier architecture discussed in the article. It provides an example of how REST Web services, Ajax, and Spring Web Flow work together to produce a desktop-like rich Web interface. The tutorial uses Jersey, Spring, MySQL, and Tomcat. It is configured and implemented in Eclipse.

This paper was made possible by research supported in part by the National Science Foundation, pursuant to its cooperative agreement with the University Corporation for Atmospheric Research . The National Center for Atmospheric Research is sponsored by the National Science Foundation. Additionally, I would like to thank Markus Stobbs from NCAR, who provided suggestions and editing for the article.

Resources

Learn

- Get more information on [Jersey](#).
- "[Architectural Styles and the Design of Network-based Software Architectures](#)" describes a framework for understanding software architecture using architectural styles and demonstrates how styles can be used to guide the architectural design of network-based application software.
- [RESTful Web Services](#) describes how you can harness the power of the Web for programmable applications.
- "[Implementing RESTful Web services in Java](#)" shows you how to write RESTful web services that conform to the JAX-RS: Java API for RESTful Web Services (JSR-311) specification.

Get products and technologies

- Read about [JSR-311](#) at the project site.
- Get more information about restlets at the [project home page](#).

About the author

Bruce Sun

Bruce Sun is a Sun Microsystems certified Java architect. He has been developing Java-based Web applications since 1998. He is currently working as a Senior Software Engineer at the National Center for Atmospheric Research (NCAR).