Chapter 1
REST and Web Services: In Theory and In Practice

Paul Adamczyk, Patrick H. Smith, Ralph E. Johnson, Munawar Hafiz

Abstract There are two competing architectural styles employed for building Web services: RESTful services and services based on the WS-* standards (also known as SOAP). These two styles have separate follower bases, but many differences between them are ideological rather than factual. In order to promote the healthy growth of Web services research and practice, it is important to distinguish arguments for implementation practices over abstract concepts represented by these styles, carefully evaluating the respective advantages of RESTful and WS-* Web services. Understanding these distinctions is especially critical for the development of enterprise systems, because in this domain, tool vendors have preferred WS-* services to the neglect of RESTful solutions. This article evaluates some of the key questions regarding the real and perceived distinctions between these two styles of Web services. It analyzes how the current tools for building RESTful Web services embody the principles of REST. Finally, it presents select open research questions to further the growth of RESTful Web services.

1.1 Introduction

Since its inception, the Web has been an open frontier of exploration in software and network system design. New ideas were tried and tested first, but organized and standardized later, once they proved their utility. For example, HTTP, the transport protocol of the Web, had been in use for more than half a decade before it was officially standardized in May 1996[1] as HTTP/1.0 [6]. However, talk about an upgraded HTTP standard started as early as in December 1995; pre-standard HTTP/1.1 [14] was adopted by Microsoft and Netscape (among others) in March 1996, and HTTP/1.1 [13] standard was completed in June 1999. The architectural principles behind HTTP and other Web standards were described by Roy Fielding in 2000 [12]. HTML has followed a similar path. It started out with a simple set

[1] The first documented version was HTTP/0.9 of 1991 [5], but it is not a true standard.

of tags for structuring text and graphics on Web pages. As the number of content types (new multimedia formats, more sophisticated ways of displaying text, interactive Web pages [17]) grew, the HTML tags were pressed into service of displaying them in various non-standard ways. After nearly two decades of this growth, new multimedia HTML tags are finally going to be added and standardized by W3C in HTML5, which is expected to be completed in 2012 [24].

A similar sequence of events – simple beginnings leading to an unruly explosion followed by some type of organization – can be observed in the realm of Web services. The first Web services were built for passing remote procedure calls (RPCs) over the Web. The idea took off quickly and resulted in a large collection of standards (beginning with SOAP and WSDL). Surprisingly, these standards were defined with little consideration to the contemporary practice; sometimes before there were any implementations to standardize. The end result of this premature standardization was confusion, rather than order that standards usually bring. In response, an alternative style of Web services, built according to the rules of the Web, began to appear. These (so-called RESTful) Web services are maturing, or, more precisely: people are re-learning to use the tried-and-true standards of the Web and applying them when building Web services. As the two styles of Web services are used side-by-side, one hopes that they will begin to have positive effects on one another. Currently, the interactions and comparisons begin to reach a constructive stage, so this is a good time to stop and reflect on the current state of affairs.

In particular, this chapter focuses on the interpretation of the widely used term, REST. Roy Fielding coined the term and codified it under four principles. In practice, people are implementing it in many ways, each harboring certain implicit conventions by the developers. Following the path of practice dictating the standards, we raise questions about the previously accepted views about REST and Web services, and identify the challenges raised by the current state of practices.

Having a standard meaning of RESTfulness would engage the enterprise community. REST has been an important part of "renegade" Web services, appealing more to independent, small-scale and "hip" developers. With concerted research effort, it would fulfill the stricter requirements of enterprise Web services; conversely, the enterprise services would benefit from its simplicity. This chapter is an important step to identify the research direction.

We begin by summarizing the theory behind RESTful Web services, and draw a comparison with WS-* services. Next we look into the usage patterns of Web services in practice: both RESTful services and WS-* services. Then we discuss some of the problems facing the existing RESTful services, how these problems make it harder to apply RESTful services to large enterprise systems, and how tools for implementing them help to alleviate these problems. We conclude by surveying some of the outstanding research problems of RESTful Web services.

Conventions used in this chapter. We consider two dominant styles of Web services: RESTful and WS-*. The term Representational State Transfer (REST) was coined by Roy Fielding to identify an architectural style based on a set of principles for designing network-based software architectures [12]. Subsequently, the term was extended to describe a style of building Web services based on the principles of REST. We use the term *RESTful* to refer to the Web services built according to this architectural style (or parts of it). We use term *WS-** to refer to services based on the SOAP standard and other WS-* standards (e.g. WS-Addressing, WS-Security), which were defined specifically for Web services

1.2 Web Services in Theory

To describe REST, we take a comparative approach and distinguish between the guiding principles of RESTful and WS-* services. We present some comparisons of concepts and technologies, and highlight research interests of both styles.

1.2.1 Principles

Roy Fielding documented REST based on the principles that emerged as the Web evolved [12]. He noticed that Web servers, clients, and intermediaries shared some principles that gave them extensibility to work on the large-scale of the Internet. He identified four principles of REST (which he called constraints) [12]:

1. Identification of resources
2. Manipulation of resources through representations
3. Self-descriptive messages
4. Hypermedia as the engine of application state (abbreviated HATEOAS)

These principles combine into a short and consistent metaphor of systems and interactions that make up the Web. The building blocks of the Web are called *resources*. A resource is anything that can be named as a target of hypertext (e.g., a file, a script, a collection of resources). In response to a request for a resource, the client receives a *representation* of that resource, which may have a different format than the resource owned by the server. Resources are manipulated via *messages* that have standard meanings; on the Web, these messages are the HTTP methods. The fourth principle means that the state of any client-server interaction is kept in the *hypermedia* they exchange, i.e., links, or URIs. Any state information is passed between the client and the server in each message, thus keeping them both stateless. It's easy to evaluate any design with such a simple metaphor. Any discrepancies will be easy to identify. However this simplicity is deceptive – if one tries to simplify it even more, bad things happen. We will discuss concrete examples of oversimplifying REST in some Web services in section 1.4.

WS-* services do not have a single metaphor. Web Services Architecture document [41] from W3C describes four architectural models of WS-*, but does not explain how they relate. One of the models is the Resource Oriented Model (which would imply REST), but as their definition of Web services suggests, the systems they consider are limited to various standards: SOAP, WSDL, and others. New capabilities are added in the form of new standards. There is no overarching description of the relationship between WS-* standards. Their definitions are constrained only by the compliance with SOAP, WSDL, and the XML schema for defining additional "stickers" in the SOAP envelope.

1.2.2 Comparison between REST and WS- Principles*

We describe 2 attempts to compare REST and WS-* services at the abstract level.

Pautasso et al. study. In the most comprehensive comparison to date, Pautasso et al. [34] compare RESTful and WS-* services on 3 levels: 1) architectural principles, 2) conceptual decisions, and 3) technology decisions.

On the level of *architectural principles*, Pautasso et al. analyze 3 principles (protocol layering, dealing with heterogeneity, and loose coupling) and note that both styles support these 3 principles. However, they can identify only one aspect common to both styles – loose coupling to location (or dynamic late binding). Consequently, they conclude that it's not possible to make a decision at this level and proceed with more detailed analysis. At the level of *conceptual decisions*, they compare 9 different decisions and find that RESTful services require the designer to make 8 of them, versus only 5 for WS-*. However, WS-* have many more alternatives than RESTful services. Finally, in the *technology* comparison, they identify 10 technologies that are relevant to both styles. In this comparison, WS-* once again offer many more alternatives than their RESTful counterparts.

Based on these results, the authors recommend using REST for ad hoc integration and using WS-* for enterprise-level application integration where transactions, reliability, and message-level security are critical.

This study illustrates two key difficulties of performing convincing comparisons of broad ideas, such as Web service styles. First, it's difficult to select the most relevant principles to compare. Second, once the principles are selected, it's difficult to identify choices that are shared by the competing ideas.

Pautasso et al do not explain why they selected protocol layering, dealing with heterogeneity, and loose coupling as the only architectural principles to compare. One would expect a comparison of principles to involve non-functional requirements [1] relevant to Web services. However, in their analysis, key -ilities (security, reliability) are only mentioned at lowest level of comparison, the technology decisions. Moreover, they shy away from comparing concepts that are relevant at the enterprise level (transactions, reliability, message-level security), even though they cite these very concepts in their concluding recommendation.

The actual comparison has two problems. First, they use the *numbers* of architectural decisions and available alternatives to choose which style is better. But counting is hardly the right metric – not every decision point has the same weight. Second, most decision points on every level have 2 options, 1 for each style, indicating that they actually have nothing in common. Only in a few cases do both styles require a decision on the same question. Nevertheless, this paper is the best-conducted comparison of principles available today. It’s unbiased, thoroughly researched, and it examines multiple points of view.

Richardson and Ruby study. A second comparison of note is presented in the book, “RESTful Web Services” [26]. The authors, Richardson and Ruby, discuss the principles that are relevant to all systems available on the Web. Even though their book is biased toward RESTful Web services, the principles they discuss would be a better starting point for making a fair comparison between the two styles.

They identify four system properties of RESTful services: 1) uniform interface, 2) addressability, 3) statelessness, and 4) connectedness. In RESTful Web services, these properties are embodied in resources, URIs, representations, and the links between them. On the contrary, WS-* services exhibits three of these four properties. Addressability and some form of connectedness are embedded in the WSDL definition of bindings and ports. Many WS-*services are stateless (although it is not an explicit requirement). Having a uniform interface shared by all services is the only property not supported by WS-*. Since these properties are relevant to both, they are a good choice for comparison.

Richardson and Ruby use a similar approach to evaluate how RESTful Web services offer capabilities which are important for enterprise-level integration. They show how to implement transactions, reliability, message-level security (concepts that Pautasso et al mention, but do not discuss) using REST. We will discuss these 3 concepts in section 1.5.1.

Both styles of Web services possess certain characteristics that guide their design and development, although they are defined in ways that make it difficult to compare them side-by-side. Next, we will look at how services are used in practice, which provides yet another perspective for comparing them.

1.3 Survey of Existing Web Services

One obstacle to studying existing Web services is the fact that many of them are not accessible to the outside world, because they are proprietary. Proprietary systems have different requirements (fewer security threats due to well known vulnerabilities, no need to adhere to common standards) that result in different choices of Web services technologies. Industry studies provide some insight about the trends in proprietary Web services, such as the planned and actual usage of Web services. One industry survey shows that the adoption of SOAP standard by enterprises increased 31% between 2002 and 2003 [7]. A follow-up survey from 2006 notes that about 12% of enterprises report completing a “full enterprise roll-out” and another 21% are in process, while 60% are still studying the feasibility of such projects [28]. Both surveys report only on SOAP Web services.

More recent results show a new trend. According to a 2008 Gartner Survey [38] there has been an increase in the number of organizations implementing Web services using Representational State Transfer (REST) and Plain Old XML (POX). RESTful Web services are considered less complex, require fewer skills and have a lower entry cost than WS-* SOAP web services. The surveyors believe that RESTful services by themselves do not provide a complete enterprise solution.

Turning our attention to public Web services, two earliest surveys of public Web services [25] [11], from 2004, discussed strictly WS-* services. Both surveys showed that some of WS-* standards (most notably SOAP and WSDL) were successfully used in practice, but they did not cover other standards. These surveys have been limited to WS-* services, perhaps unwillingly, because they considered the presence of a WSDL file as a necessary prerequisite of a valid Web service.

In order to build on their work, we have studied various Web services repositories (including the only extant ones cited by these surveys) to analyze the available public Web services from the perspective of architectural styles they follow. We performed these surveys in mid-2007 and again in mid-2010 by examining the Web services listed in the following repositories:

- `xmethods.net`
- `webservicex.net`
- `webservicelist.com`
- `programmableweb.com`

These repositories describe only publicly accessible Web services. While SOAP services are easy to find automatically (by checking for the presence of the WSDL file), RESTful services are documented in non-standard ways that make their automatic discovery impossible. We examined the type of each service manually, by reading its documentation. We have identified five mutually exclusive categories of Web service styles: RESTful, WS-*, XML-RPC, JavaScript/AJAX, and Other. XML-RPC was the first attempt at encoding RPC calls in XML (which later evolved into SOAP). AJAX (Asynchronous Javascript and XML) is used for interactive services, e.g. maps. The Other category groups many other types of services, including RSS feeds, Atom, XMPP, GData, mail transfer protocols. The most popular styles of Web services in each repository are shown in Table 1.

Table 1.1 Web service styles used in public services. Survey conducted in 2007 and 2010. Some service are available in two or more styles. The number of unique services is shown in parentheses.

Style	**xmethods**		**webservicex**		**webservicelist**		**programmableweb**	
	2007	**2010**	**2007**	**2010**	**2007**	**2010**	**2007**	**2010**
RESTful	3	0	0	0	103	144	180	1627
WS-*	514	382	71	70	233	259	101	368
XML-RPC	1	0	0	0	6	21	24	53
Javascript/AJAX	0	0	0	0	0	9	30	130
Other	0	0	0	0	98	26	60	77
Total (Unique)	514	382	71	70	430 (411)	459 (386)	395 (340)	2255 (2179)

At a first glance, these results could not possibly paint a more inconsistent picture. Each repository shows different trends. However, the differences arise from the nature/focus of these repositories. The first two repositories, which list (almost) exclusively WS-* services, advertise services that require payment for access. The second repository appears to be closed to registration (we could not find any way to contact the owners to register a new service) which may imply that they are advertising only the services which they own. The numbers of services listed in these two repositories have not changed much in the last 3 years.

The latter two repositories feature a variety of Web service styles, with RESTful and WS-* services being the two most popular styles in both the 2007 and 2010 tally. Programmableweb.com is the only repository that shows an increase in the number of services; a 5-fold increase over the observed period. Its data shows increase in all types of services, but mostly in RESTful ones, which currently account for about 75% of services listed, compared to less than 50% 3 years ago.

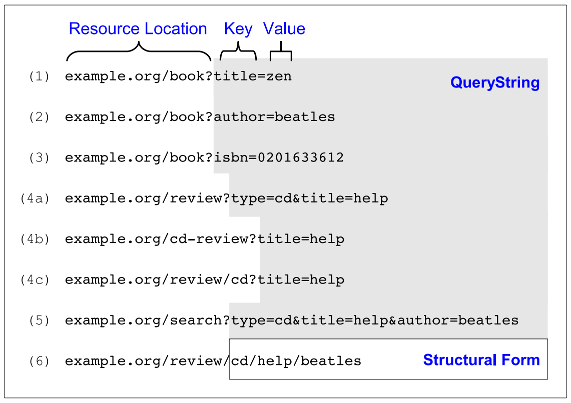
These results, although insufficient to determine conclusively which style is more popular (and why), indicate that a wide variety of public Web services is in use and that a sizeable number of RESTful services has been created recently, even if not all of them are widely known.

1.4 REST Concepts in Practice

With so many public Web services to analyze, we were able to identify many trends in how closely services follow the theoretical principles of REST. SOAP principles are encoded in XML-based standards that are easy to enforce by tools. The designer selects the necessary features (standards), then finds the tool that supports them. The actual development is easy. But since this book is about REST, we will focus on RESTful Web services, and bring up WS-* occasionally to compare and contrast specific features.

REST principles are more like guidelines, so nothing obviously bad happens if one breaks them – an occasional scorn of peers, sub-par performance, or awkward API definitions that require constant updates. In this section we will review how REST principles are embodied and implemented in actual RESTful Web services.

According to the principles of REST, which we introduced in section 2, every resource is identified with a URI. In response to HTTP messages, resources return their representations to clients, or the clients modify the resources. Proponents of RESTful Web services typically say that every service needs to follow the CRUD model, which defines one method for creating, reading, updating, and deleting a resource on the server (corresponding to PUT, GET, POST, and DELETE methods). This approach enables invoking different operations on a resource by applying a different HTTP method. This is only possible if resources are defined in a correct way. Figure 1.1 shows some examples of valid URI definitions of resources. All resources except for example 5 can be accessed with any of the CRUD methods and return intuitive results.



```
(1)  example.org/book?title=zen
(2)  example.org/book?author=beatles
(3)  example.org/book?isbn=0201633612
(4a) example.org/review?type=cd&title=help
(4b) example.org/cd-review?title=help
(4c) example.org/review/cd?title=help
(5)  example.org/search?type=cd&title=help&author=beatles
(6)  example.org/review/cd/help/beatles
```

Fig. 1.1 Examples of RESTful hypermedia defined as URIs. Examples 1-5 use query strings. Examples 4a-c show alternative ways to define the same resource. Example 5 is a bad resource definition, because "search" does not represent a single resource; it's the entire Web service. Example 6 uses the structural form instead of query strings.

Before giving a detailed review of violations and misunderstandings, we would like to point out one good example of a Web service that has proper RESTful design, Amazon S3 (Simple Storage Service) [3]. S3 defines true resources and uses HTTP methods (GET, PUT, POST, DELETE, even HEAD) for manipulating them. It uses HTTP error codes correctly and shows how to map various errors to HTTP codes (the API references 13 unique HTTP status codes in the 300-500 range). S3 also supports caching by including ETag header that clients can use in conditional GET. This effort is even more commendable, because this service also defines a WS-* version, so it would be tempting to make the RESTful version by directly translating the WSDL to pseudo-resources.

However, most RESTful services are not designed as diligently. They neglect to follow the principles in various ways. We will look at some representative mistakes by revisiting the 4 principles of REST.

1.4.1 Identification of Resources

Every designer of a RESTful service must answer the question: What constitutes the resources of the system? Ideally, any concept within the system that has a representation should be exposed as a resource. RESTful APIs ought to map services to resources, not to the internal implementation of services. But many existing services do this wrong.

The most blatant violation of the first principle of REST is a Web service that defines only one resource. Let's consider, for example, the Digg Web service. In order to access the news, a user is required to invoke the following URI:

`http://services.digg.com/2.0/user.getMyNews`

Note that this URI is similar in structure to example 5 in Figure 1. The Web service defines a single resource `services.digg.com/2.0/` and all specific requests are encoded as a string representing a method call (`getMyNews`) on an object (`user`). To translate this URI into an internal request, the Web service extracts the string element after the last "/" and uses it to invoke an internal function whose name is hard-coded in the request URI. This URI definition poses a problem, because it's not possible to call HTTP methods on such a resource. In order to manipulate the contents of the user object, new resources for adding, updating, and deleting the "news" must be defined.

Defining resources is hard. Consider, for example, a hypothetical Web service that provides information about books and music. It should define multiple resources, `book`, `cd`, `review` that are queried by title, author, or ISBN. Example 1 in Figure 1.1, `example.org/book?title=zen` represents a resource for finds books containing "zen" in the title. Examples 2 and 3 show how to query the resource by author and ISBN. It's more difficult to define a resource corresponding to review. The system could have one review resource (as in 4a), a dedicated resource for each product type (in 4b), or a composite resource (review) with individual children resources, one per product type (4c). They all are valid choices. Alternatively, URIs can use "/" instead of "?" to separate parameters, as in Example 6. Note that this format requires implicit understanding of the structure of this URI, which is not defined in the URI.

The problem of identifying resources is similar to teaching object-oriented design to programmers, who were first taught procedural languages - it requires a changed mindset. In object-oriented design, the first step is to define "things" or "nouns" as objects/classes. In the second step, the public methods of the object are defined. In any non-trivial problem, these two steps identify many objects and many methods. The application is built by connecting the objects, which invoke methods on one another. A similar approach can be applied to defining resources, except that only the first step identifies many objects (i.e. resources). The available HTTP methods are defined in the standard and links between resources are traversed at run-time. Thus steps 2 and 3 come for free in HTTP, but only if step 1 is done well.

1.4.2 Representations

If resources support multiple representations, they can produce responses in different data formats. In HTTP, clients specify their preferred formats in Accept-* headers for content negotiation. By confirming to HTTP, RESTful Web services can support multiple types of response (MIME) formats, just like the Web does, which makes it easy to comply with this principle.

Many RESTful Web services support at least two response formats (typically XML and JSON). Library of Congress Subject Headings Web service is the only service listed at programmableweb.com that advertises the support of content negotiation. It serves content in four different types (XHTML with embedded RDFa, JSON, RDF/XML, and N3). Unfortunately, other services do not appear to support this important feature of HTTP, because we did not find it in their documentation.

1.4.3 Self-descriptive Messages

REST constrains messages exchanged by components to have self-descriptive (i.e. standard) definitions in order to support processing of interactions by intermediaries (proxies, gateways). Even though HTTP/1.1 defines 8 methods, only two of them, GET and POST, have been used extensively on the Web, in part because these were the only methods supported by Web browsers.

Early Web services were having difficulties understanding the differences between even these two methods. [2] Some services defined GET for sending all requests to resources, even if the requests had side effects. For example, initially, Bloglines, Flickr, and Delicious Web services defined GET for making updates to these services [8]. Other services specified that clients can use GET and POST interchangeably, which is obviously wrong. Consequently, these services were misusing Web proxies and caches polluting them with non-cacheable content, because these Web systems rely on standard meanings of HTTP methods. Since then, the offending APIs were modified, but the underlying problem of understanding the semantics of HTTP methods still remains.

One reason for this difficulty is that the 4 CRUD methods are not sufficient to express all operations cleanly. CRUD covers basic operations, which need to be combined into sequences in order to implement even the simplest[3] transaction [26]. That's why many RESTful services try to encode their operations into URIs in RPC style even though they know that it violates REST. Another reason is the CRUD metaphor. The difference between POST and PUT is not obvious. They don't map exactly to CRUD's "create" and "update" respectively. PUT carries a representation produced by the client, which the server should use to replace its contents (so it serves as both create and update). POST means the server decides how to use the representation submitted by the client in order to update its resource.

This problem of not taking full advantage of HTTP methods is not unique to Web services. Typical Web applications (accessible via browsers) use only two HTTP methods in practice. In a study of HTTP compliance of Web servers [2], we found that Web servers and intermediaries understand correctly only GET and POST methods. Only a fraction of popular websites send compliant responses to other HTTP methods, even though the popular Web servers implement all these methods correctly. These compliance results haven't changed much since HTTP/1.1 standard has been released, in 1999. This doesn't bode well for any services that depend on Web browsers, e.g. for testing.

[2] GET sends data from the server to the client, in the response. POST sends the data from the client to the server, in the request. From the server's perspective, GET is for reading, and POST is for writing.

[3] A simple bank transaction, e.g. transferring $100 from savings to checking, involves 3 HTTP methods. First, create a resource for the transfer using POST. Next, send a PUT to the resource specifying the withdrawal of $100 from savings. Finally, send a second PUT to deposit $100 to checking. Note that the client needs to verify the success of each step. If a step fails, the client needs to send a DELETE to the transaction resource to abort the transaction.

We believe that it's not necessary to insist on the use of CRUD methods for Web services. It is not possible to express every operation with these methods anyway. The principle of self-describing messages means that methods should be used according to their standard definitions. Case in point is the new HTTP method, PATCH, added in March 2010 [10]. It is intended to replace POST by providing with more precise semantics for making updates on the server. With POST, the client cannot specify *how* the resource is to be updated. Unfortunately, the definition of PATCH does not define the structure for including the instructions to update (ei.e patch) the resource. A standard definition of the instructions will be necessary to make this method interoperable. As the additions of PATCH indicates, the set of *relevant* HTTP methods is not static. The WebDAV protocol (which RESTful proponents tend to overlook it) defines 8 more methods for distributed authoring and manipulating collections of resources [18]. Thus RESTful Web services have many self-describing methods to choose from. Although today most Web services don't use their HTTP methods right, we hope that in time they will.

1.4.4 HATEOAS

Hypermedia as the engine of application state means that neither client nor server needs to keep the state of the exchange in a session, because all the necessary information is stored in the exchanged HTTP messages (in the URI and the accompanying HTTP headers). Defining self-contained links is critical for RESTful Web services, because these links make it possible to traverse, discover, and connect to other services and applications.

However this is difficult, because complex interactions translate to complex URIs. Complex applications have many states that the client needs to be aware of. HATEOAS forces Web services to expose the states as links, which duplicates the internal design of the service. That's why many RESTful Web services resort to exposing the underlying API of the service even if they know it's wrong.

Many services require the client to send user-specific information (e.g. user-id) in every request URI. As a result, the same requests from two different clients appear unique to the Web caches, because caches use URIs as keys for the data. Sending user-specific information is often unnecessary (especially when the user sends a generic query), but it's used extensively by Web services providers to limit the number of accesses from each client. Since HTTP caching cannot be used in this case, the service must handle more requests, which defeats the purpose of rate limiting. This seemingly innocuous (but often occurring) lapse violates two principles – the identification of resources and HATEOAS; it also affects cacheability.

1.4.5 Other important concepts

The HTTP standard defines the meaning of different error conditions and several mechanisms for caching. Compliant RESTful Web services should follow them.

Initially, RESTful services copied their error-handling mechanism from SOAP. Many Web services would not use HTTP status codes (e.g. "404 Not Found") to describe the result of a request, but rather always returns "200 OK" with the actual status is hidden in the response body. Other services (e.g. earlier versions of Yahoo Web services) defined their own status codes that were incompatible with the standard ones. By using service-specific codes, they would not take advantage of existing Web systems that understand these codes thus forcing clients to build specialized, non-interoperable software to handle them. Fortunately, most Web services we surveyed now do use HTTP status codes, and only add service-specific extensions for new statuses. For example, Delicious uses codes 500 and 999 to indicate that user request was throttled (because of exceeding a pre-defined limit of connections). HTTP does not have a status that corresponds to this condition, so it makes sense to define a new one.

Our survey gathered little information about caching. Aside from exceptional Web services like S3 (and even they don't use the term caching in the documentation), RESTful services do not document if they support caching. Of course, the ones that employ user-ids could not benefit from caching anyway.

As the length of this section indicates, RESTful services still have difficulty in following the principles of REST. There are few fully compliant service definitions, but it's easy to find examples of services that violate any of the principles. On the bright side, we have observed a lot of improvements in compliance over the last few years. RESTful services, by the virtue of being public are more open to general scrutiny. Users can discuss the design decisions in the open, criticize them, and see changed in the next version. To gain a better perspective of the positive changes, the reader is encouraged to browse the discussion of these and other violations documented few years ago at RESTwiki [37].

An important question is: Why are many services that attempt to be RESTful not compliant with the principles of REST? Are these principles too restrictive? too hard to implement? Unnecessary for Web services (as opposed to Web pages/applications)? It's still too early to tell.

1.5 Frameworks for Building RESTful Web Services

The improvements in the understanding of the principles of REST, as indicated by the slow but steady elimination of bad design decisions from public RESTful Web services can be attributed to software tools and frameworks that have began to appear in the last few years.

1.5.1 Support of REST Principles

Many frameworks and tools for building RESTful Web services are available today. They are written in different programming languages and range from simple to quite sophisticated in their support of HTTP and other Web technologies. As they continue to improve, misunderstandings and violations present in today's Web services will likely lessen.

We have examined 10 popular frameworks that provide automated support for building software according to the principles of REST. Some frameworks, like Ruby on Rails and Spring are generic Web frameworks, while others are specific to building RESTful services. Table 1.2 summarizes key features of these frameworks, grouped by REST principles. The frameworks are listed alphabetically, sorted by the programming language and name. The second column in the table shows how these frameworks support building resources (corresponding to REST principles 1 and 4). Almost all of these frameworks provide some support for defining resources (URIs) and hyperlinks – through URI templates [19], annotations in the target programming language, or other types of mappings. The third column shows which types of multimedia are supported and how (principle 3). Most frameworks enable generation of multiple representation formats. The fourth column shows which HTTP methods are supported (principle 4). Most of them support the 4 (CRUD) methods, directly, or by specifying the desired method in an auxiliary parameter (such as the `X-HTTP-Method_Override` header, or the hidden "_method" form field [26]). The last column points out other interesting features provided by the frameworks. Few brave frameworks have ventured into implementing more advanced concepts of caching, automated testing, or authentication.

Several of the Java frameworks support JAX-RS, a Java API for RESTful Web services. They are Jersey (considered the reference implementation), Restlet, and RESTEasy. JAX-RS specifies how to map Java classes to Web resources using Java annotations. The annotations specify the relative path of the resource (part of the URI) for a Java class, which Java methods correspond to HTTP methods, which media types are accepted by the class, and how to map class properties to selected HTTP headers [22].

Aside from Django, all the Python and PHP frameworks offer only rudimentary support for REST. Other frameworks include more advanced features, but they still fall short of supporting all principles of REST. Most frameworks define schemes for mapping URIs to classes and methods, but not all of them are as flexible as HTTP requires, e.g. Ruby on Rails imposes limits on URI formats. Only one framework (Restlet) supports all HTTP status codes. No framework supports all flavors of HTTP caching, and many do not support caching at all.

The principle of HATEOAS (unambiguous semantics for following and embedding links) is not well supported. Only the RESTfulie framework emphasizes the importance of this principle. Let's consider a simple example of the expected behavior. When a client requests a resource (e.g. information about a collection of items) it should be easy to construct a URI to refer to an individual item from that collection. Frameworks should provide built-in support for such conversions of URIs.

Table 1.2 How frameworks for building RESTful Web services support the principles of REST

Name (Prog. Language)	Resources and HATEOAS	Representation	Messages	Other (API, caching, status codes, etc)
Jersey (Java)	Annotations for URI mappings	MIME types, XML, JSON and Atom	CRUD	Support for JAX-RS. Testing framework.
RESTEasy (Java)	Annotations for URI translations and variable mapping	Annotations for output representations (many types supported). Content negotiation	CRUD	Output caching and compression Support for JAX-RS
Restlet (Java)	URI templates [19] and variable binding	Supports various output representations	CRUD	Support for JAX-RS. Caching headers set in `Conditions` class. Security checks added via filters. All HTTP status codes.
Spring (Java)	Templated URIs using Java annotations	Content negotiation with Accept header or by URL inspection (read file extension)	GET and POST directly, PUT and DELETE with `_method`	ETag header for caching.
Recess (PHP)	URI templates [19] and variable extraction using annotations	(–)	CRUD	(–)
Routes (Python)	Proper URL syntax; No IDs in query parameters	(–)	CRUD	(–)
CherryPy (Python)	Simple mapping: HTML forms to Python variables	HTML forms	GET and POST	An object tree is generated which maps requests to functions
Django (Python)	URI templates for mapping advanced URL patterns to Python code	Targeted output formats: XML, JSON, YAML	CRUD	Caching. HTTP status codes supported by Python libraries
RESTfulie (Ruby, Java)	emphasizes hypermedia links	many formats; content negotiation	CRUD	HTTP status codes; integrates with Ruby on Rails
Ruby on Rails (Ruby)	Route configs map URI to `Component` class (imposes URI conventions)	Excellent support of many data formats – e.g. Accept header	CRUD	Conditional GET for caching RESTful authentication

Currently, this mapping work must be implemented in the client code, because most frameworks do not support it.

Overall, the RESTful frameworks need to include more functionality to be fully compliant with REST. But the biggest problem is that even if they do implement the support for a principle, the frameworks have no mechanisms to enforce that it is applied correctly in the client code.

1.5.2 Ready for the Enterprise?

Frameworks make it possible to build bigger Web services, and their capabilities keep on growing. Is that enough to persuade enterprise system architects to switch to RESTful Web services? Recall the study of Web services by Pautasso et al we discussed in section 2 [34]. They cite security, reliable messaging and transactions as key differentiators between RESTful and WS-* services. To be ready for enterprise, RESTful frameworks need to support these features. Richardson and Ruby show in [26] how these concepts can be implemented using HTTP.

For basic message-level security, it's enough to use HTTPS. But more complex capabilities such as signatures, encryption, or federation (enabling a third party to broker trust of identities) cannot be supplied by HTTP alone. Further research is required to define these concepts properly in RESTful Web services (more about this in section 1.6.)

To provide reliable messaging, one needs to ensure that all HTTP methods are idempotent. This property makes it possible to replay any method, as necessary, to make sure that it succeeded. Of course, this approach to reliable messaging is tedious and currently requires a lot of manual coding on the client side.

Implementing transactions with CRUD messages requires exchanging many HTTP messages, which can get complex quickly (as we saw in section 4.3). Current frameworks are not mature enough to abstract out/encapsulate common transaction patterns. But transactions are needed as building blocks of workflows, which occur often in enterprise systems. A proposed extension to the Jersey framework introduces *action resources* for specifying workflows [21]. Each action resource exposes one workflow operation available on the service. The client obtains the workflow specification (i.e. the list of action resources) at the beginning of the sequence. In line with the principle of HATEAOS, it's the client's responsibility to keep track of the current state of the system throughout the execution in order to invoke the workflow resources in the correct order. This is a dynamic approach, because the exact sequence of the workflow need not be specified until the client begins to execute it.

But even if security, reliable messaging, and transactions are solved successfully, RESTful services must also ensure scalability. Current RESTful services are small, and the services that are true to REST are the smallest among them. Amazon S3 has 2 top-level resources and 10 sub-resources. RESTify Day Trader, another well-defined Web service, has only 6 resources. Compared to large legacy systems WS-* services hide, most RESTful services are toy-sized. Annotations, the common solution used in today's frameworks simplify design, but they don't scale.

Today's frameworks are not yet ready to support enterprise needs. They do not implement advanced security features or transactions; they do not verify that HTTP methods they generate are idempotent, which is the necessary prerequisite for reliable messaging; they are not scalable. Implementing these features is a matter of time, because HTTP already defines most of the necessary concepts to perform these tasks. However, it's not enough that the frameworks implement the necessary functionality. The frameworks must guide and force the users to recognize the correct features for the job and to apply them correctly.

1.6 Open Research Problems of RESTful Services

REST originated at the intersection of academia and software development, among the architects of the World Wide Web. Fielding's research culminated in authoritative versions of HTTP and URI standards that define the unique characteristics of the Web. Unfortunately, researchers have only recently started to work on RESTful services. As late as 2007, there were no papers about RESTful Web services in either ICWS, ECOWS, or WWW conferences. In 2010, ICWS has featured several papers about RESTful services and the WWW conference has hosted the first "Workshop on RESTful Design (WS-REST 2010)" [33], which is a welcome sign.

Proponents of RESTful Web services made their first attempts to reach the research community via conference presentations [35] [20], and computer magazine editorials [39]. Recently, survey papers [34], and new research work [33] [32] began to appear. Hopefully, this book will advance the research even farther.

The problems we discuss below are mostly concerned with non-functional requirements and how they can be supported by RESTful services. Many of these research efforts are defining new Web standards. HTTP linking [30] aims to improve cache invalidation. HTTP PATCH [10] defines a new method to make more maintainable services. URI templates [19] make it easier to define groups of resources with regular expressions. OAuth [23] secures authentication and data sharing in a HTTP-based system. We discuss some of these efforts.

1.6.1 Caching

Of many aspects of performance, caching is one of the best examples of why it pays to use HTTP correctly. The data may be cached by the client, by the server, or by intermediaries, such as Web proxies. In the early days of mostly static content, 24-45% of typical Web traffic was cacheable [9]. Today, the estimated range is 20-30% [27], which is very impressive considering how dynamic the Web content is.

Unfortunately, most of the RESTful services aren't benefitting from caching: many frameworks don't support caching, and typical URIs are not cache-friendly, because RESTful Web services require user info in each request. We have already discussed how user-ids are used for rate-limiting, in section 1.4.4. It is unlikely that Web services will ever change this policy. Instead, it would be better to move user-specific information out of the URIs, so that the responses can still be cacheable.

An upcoming addition of HTTP Linking [30] (for improving cache invalidation) indicates that the HTTP community values caching. However, it's very difficult to keep up with all the variations: caching headers, tags, expirations, and conditional methods. Caching is so complex that even the upcoming HTTPbis specification from IETF divides this topic into two documents (Caching proper and Conditional Requests). Caches are not unique to the Web: caching in computer architecture is understood well. We are lacking a single, consistent model of caching on the Web.

1.6.2 Maintainability

Typical maintenance tasks of Web services (adding new features, fixing service APIs) affect services themselves, their documentation, the client code, and even the development tools. Since RESTful Web services are still prone to wholesale changes, each of these facets offers ample opportunities for research.

For example, it would be easier to develop RESTful frameworks if they were more RESTful themselves. A possible alternative to using annotations in RESTful frameworks is the Atom Publishing Protocol (APP) [31]. Atom is an XML vocabulary for describing publishing semantics; it's a popular syndication feed format. APP defines representations of resources by extending Atom. APP specifies how to access resources via HTTP methods and it supports adding extensions to the APP specification. Unlike annotations, APP is built using REST concepts. It could be used in RESTful frameworks to specify associations between client code and HTTP constructs at higher levels of abstraction.

1.6.3 Security and Privacy

Securing RESTful Web services is a multi-faceted endeavor: it involves securing the data, as well as the entire communication. One must protect the confidentiality and integrity of data. The data in transit should be filtered for malicious payload. The communication should support authentication and access control, and ensure that the privacy of the communicating parties is not compromised.

Compared to the WS-Security framework [40], RESTful services rely on various add-ons that work on top of HTTP. HTTPS [36] is widely used for confidentiality, but it only provides hop-by-hop security. Developers should adopt message level security mechanisms. Unlike WS-*, there are no standards to follow, but practitioners follow various reference architectures, e.g. Amazon S3 service [3]. Amazon S3 also incorporates timestamps to guard against request replaying. Various client side and server side filters should be employed to validate the content.

HTTP supports basic and digest-based authentication mechanisms [16], but both have their weaknesses [4]. Current services delegate identity management and authentication mechanism to a third party, and rely on a claims-based authentication model. Technologies for supporting authentication for HTTP-based services are emerging, e.g. OpenId [15] for federated identity, and OAuth 1.0 [23] for authentication and data sharing. These protocols open up new avenues of research. For example, OAuth is going through a revision in October 2010, where the protocol writers are considering dropping cryptographic operations and relying on SSL to protect plaintext exchange of authentication tokens. They are trading off security for ease of programming, but this decision should be validated by research. Another emerging protocol is XAuth [29], an open platform for extending authenticated user services across the web, which has a lot of open security problems.

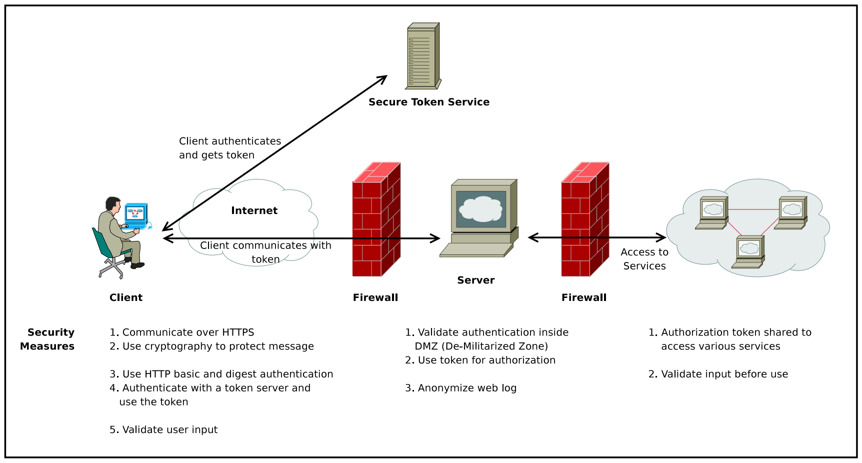

Secure Token Service
Client authenticates and gets token
Internet
Client communicates with token
Client
Firewall
Server
Firewall
Access to Services
Security Measures
1. Communicate over HTTPS
2. Use cryptography to protect message
3. Use HTTP basic and digest authentication
4. Authenticate with a token server and use the token
5. Validate user input
1. Validate authentication inside DMZ (De-Militarized Zone)
2. Use token for authorization
3. Anonymize web log
1. Authorization token shared to access various services
2. Validate input before use

Fig. 1.2 Security measures adopted at different layers in RESTful systems

Storing RESTful URIs in web logs may lead to privacy problems if the logs are not protected and anonymized. WS-* services do not store sensitive data in HTTP method signature and query strings. On the other hand, RESTful URIs become the audit trail, and they should be anonymized.

Figure 1.2 describes a hypothetical model of how the security and privacy measures can be applied together. It shows a secure token service, a key entity in a third party authentication model. Note that the figure does not define the actual steps of an ideal protocol; it is an open research problem. Researchers also need to figure out how the security measures fit the REST model.

1.6.4 QoS

When multiple providers offer the same service, a client has a choice and can select the most suitable one. Often, this choice comes down to the Quality of Service (QoS) parameters. RESTful Web services today ignore QoS requirements; their only concern is providing functional interfaces. To add QoS parameters to RESTful services, a language for describing the parameters and a mechanism to incorporate the description in the HTTP payload is needed. Defining a standard QoS description language might benefit from the work in Semantic Web. Semantic Web ontologies define standard ways of interpreting information, such as QoS parameters, enabling all clients to interpret them the same way.

1.6.5 *Studies of Existing Systems*

Web services are good candidates for studying software engineering concepts of large, publicly available systems. But there have been few successful studies of RESTful services, or side-by-side comparisons of a service that exposes two interfaces defined in the competing styles (one RESTful, one WS-*).

It is not easy to compare these two styles at the level of principles. The first order of research is to identify good principles for making the comparison. Zarras [42] identifies the following principles for comparing middleware infrastructures: openness, scalability, performance, distribution transparency. Properties of software architectures [1] is another source of principles to consider. Another possibility is to re-apply the principled approach, used by Fielding to derive REST, in order to define both RESTful and WS-* architectural styles. This would entail applying additional constraints, one at a time, to derive a complete definition of an architectural style.

1.7 Conclusion

RESTful Web services (and Web services in general) pose the first big test of the principles of REST, as identified by Fielding. Even though RESTful services might look like typical Web system, they aren't, and WS-* services are clearly different.

Up until a few years ago, there was a simple dichotomy between REST and WS-*. RESTful services were used only for simple, public services. In contrast, enterprise standards, tools vendors, and the research community were only concerned with WS-* services. This is no longer the case – both styles are being used in all domains. The new challenge is to use them right, and to be able to align them to solve the real problems of the enterprise. Can RESTful services scale up to the enterprise-size challenges? We believe so. Amazon, Google, Yahoo, Microsoft, and other big companies have been building enormous scalable and extensible systems on the Web. RESTful services have the same basic principles to follow.

This concludes our whirlwind overview. Other chapter in this book will explore these topics in more details.

References

1. L. Bass and P. Clementes and R. Kazman. *Software Architecture in Practice, 2nd Edition*. Addison Wesley, 2002.
2. P. Adamczyk, M. Hafiz, and R. Johnson. Non-compliant and Proud: A Case Study of HTTP Compliance, DCS-R-2935. Technical report, University of Illinois, 2007.
3. Amazon. Amazon Simple Storage Service API Reference.
4. Apache HTTP Server v2.2. Authentication, authorization and access control.
5. T. Berners-Lee. The original HTTP as defined in 1991, 1991.
6. T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0, May 1996.

7. J. Correia and M. Cantara. Gertner sheds light on developer opps in web services. *Integration Developers News*, June 2003.
8. Dare Obsanjo Blog. Misunderstanding REST: A look at the Bloglines, del.icio.us and Flickr APIs.
9. B. M. Duska, D. Marwood, and M. J. Freeley. The measured access characteristics of World-Wide-Web client proxy caches. In *USENIX Symposium on Internet Technologies and Systems, USITS*, 1997.
10. L. Dusseault and J. Snell. RFC 5789: PATCH Method for HTTP, Mar. 2010.
11. J. Fan and S. Kambhampati. A Snapshot of Public Web Services. In *SIGMOD Record, Vol. 34, No. 1*, Mar. 2005.
12. R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. Technical report, University of California, Irvine, 2000.
13. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol — HTTP/1.1, June 1999.
14. R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk Nielsen, and T. Berners-Lee. RFC 2068: Hypertext Transfer Protocol — HTTP/1.1, Nov. 1998.
15. B. Fitzpatrick. OpenID, 2005.
16. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. RFC 2617: HTTP Authentication: Basic and Digest Access Authentication, June 1999.
17. J. Garrett. Ajax: A new approach to web applications, Feb. 2005.
18. Y. Goland, E. J. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring WebDAV. Internet proposed standard RFC 2518, Feb. 1999.
19. J. Gregorio, R. Fielding, M. Hadley, and M. Nottingham. URI Template (draft), Mar. 2010.
20. H. Haas. Reconciling Web services and REST services (Keynote Address). In *3rd IEEE European Conference on Web Services (ECOWS 2005)*, Nov. 2005.
21. M. Hadley, S. Pericas-Geertsen, and P. Sandoz. Exploring Hypermedia Support in Jersey. In *WS-REST 2010*, Apr. 2010.
22. M. Hadley and P. Sandoz. JAX-RS: Java API for RESTful Web Services (version 1.1), Sept. 2009.
23. E. Hammer-Lahav. RFC 5849: The OAuth 1.0 Protocol, Apr. 2010.
24. I. Hickson. HTML5: A vocabulary and associated APIs for HTML and XHTML, Oct. 2010.
25. S. M. Kim and M. Rosu. A Survey of Public Web Services. In *WWW 2004*, 2004.
26. L. Richardson and S. Ruby. *RESTful Web Services*. OReilly, Oct. 2007.
27. M. Nottingham. HTTP Status Report. In *QCon*, Apr. 2009.
28. J. McKendrick. Service Oriented Blog.
29. Meebo Dev Blog. Introducing XAuth, Apr. 2010.
30. M. Nottingham. Web Linking (draft), May 2010.
31. M. Nottingham and R. Sayre, Ed. RFC 4287: The Atom Syndication Format, Dec. 2005.
32. H. Overdick. Towards resource-oriented BPEL. In C. Pautasso and T. Gschwind, editors, *WEWST*, volume 313. CEUR-WS.org, 2007.
33. C. Pautasso, E. Wilde, and A. Marinos. First International Workshop on RESTful Design (WS-REST 2010), Apr. 2010.
34. C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. ”big”’ web services: making the right architectural decision. In *WWW ’08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM.
35. P. Prescod. Roots of the REST/SOAP Debate. In *Extreme Markup Languages, EML*, 2002.
36. E. Rescorla. RFC 2818: HTTP over TLS, May 2000.
37. RESTWiki. http://rest.blueoxen.net/cgi-bin/wiki.pl.
38. D. Sholler. 2008 SOA User Survey: Adoption Trends and Characteristics. Technical Report G00161125, Gartner, Sept. 2008.
39. S. Vinoski. Serendipitous reuse. *IEEE Internet Computing*, 12(1):84–87, 2008.
40. Web Service Security (WSS). Web Services Security: SOAP Message Security 1.1. Technical report, OASIS, Feb 2006.
41. W. working group note. Web Services Architecture. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.
42. A. Zarras. A comparison framework for middleware infrastructures. *Journal of Object Technology*, 3(5):103–123, 2004.