Approved for Public Release; Distribution Unlimited
Case # 10-1691

MTR100093
MITRE TECHNICAL REPORT

RESTful Services Guidance for Developers v 1.0

G. Beuchelt
T. Kehoe
P. J. Miller
R. Modeen
C. Partridge
M. Patron
D. P. Robbins
R. O. Wilson

April 2010

MTR100093
MITRE TECHNICAL REPORT

RESTful Services Guidance for Developers v 1.0

G. Beuchelt
T. Kehoe
P. J. Miller
R. Modeen
C. Partridge
M. Patron
D. P. Robbins
R. O. Wilson

April 2010

Sponsor: ESC
Dept. No.: E111
Contract No.: FA8721-09-C-0002
Project No.: 03101CF0-AA

The views, opinions and/or findings contained in this report are those of The MITRE Corporation and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

©2010 The MITRE Corporation.
All Rights Reserved.

Approved for public release;
distribution unlimited.

Abstract

Representative State Transfer (REST) is an architectural pattern that explains the technical underpinnings responsible for the tremendous success of the World Wide Web. The REST pattern and supporting technologies not only support human focused web browser operations but also machine-to-machine information exchanges. In this document we focus on the latter. REST is less complicated than other approaches, easy for developers and users to understand, and easy to implement. Further, the approach is scalable to large enterprises due to a fundamental tenet of REST: stateless interactions. The same aspects of REST which drive companies such as Amazon and Google to use REST to deliver capability to their users make it an attractive and useful technology for the Department of Defense (DoD).

The goal of this document is to introduce the REST pattern and to share lessons learned gathered through our own development efforts using REST for the DoD, and the study of current commercial practices. We cover both the REST concept in general and the supporting technologies needed to employ REST effectively for developing web services. The authors do not consider this document to be an authoritative mandate, but instead an informational snapshot of current practices for REST services that includes areas which are still evolving such as service security.

Release of V1.0 is intended to serve as the vehicle for further review by a more extensive developer community. As required, revised versions of this document will be published.

This Page Intentionally Blank

Table of Contents

List of Tables

This Page Intentionally Blank

1 Introduction

Introduction

Representative State Transfer (REST) is an architectural pattern that explains the technical underpinnings responsible for the tremendous success of the World Wide Web. The REST pattern and supporting technologies not only support human focused web browser operations but also machine-to-machine information exchanges. In this document we focus on the latter. REST is less complicated than other approaches, easy for developers and users to understand, and easy to implement. Further, the approach is scalable to large enterprises due to a fundamental tenet of REST: stateless interactions. The same aspects of REST, which drive companies such as Amazon and Google to use REST to deliver capability to their users, make it an attractive and useful technology for the Department of Defense (DoD).

The goal of this document is to introduce the REST pattern and to share lessons learned gathered through our own development efforts using REST for the DoD and study of current commercial practices. We cover both the REST concept in general and the supporting technologies needed to employ REST effectively for developing web services. The authors do not consider this document to be an authoritative mandate, but instead an informational snapshot of current practices for REST services that includes areas which are still evolving such as service security.

This Page Intentionally Blank

2 Service Styles and Usage Patterns

REST is an architectural pattern that explains the technical underpinnings responsible for the tremendous success of the World Wide Web. The term Representational State Transfer was coined by Roy Fielding in his 2000[1] Doctoral Dissertation in which he explores the basic structure of the Web. REST does not describe specific protocols, data formats, or a sequence for interaction that is typical for most specifications. REST does describe the architectural principles and components that enable the World Wide Web to function successfully.

2.1 Key Elements of RESTful Services

2.1.1 Resource

REST is built on the concept of an information resource. An information resource is any piece of information or content that is uniquely identifiable and can be expressed in a textual or binary representation transmittable over a network. In the past a resource was regarded as the data intended for viewing within a browser, such as a HyperText Markup Language (HTML) page with embedded images and video clips. However, as the Web evolved, software consumers of web services emerged so that now the concept of a web resource or web content can just as often refer to data objects consumable by software applications.

2.1.2 Uniform Resource Locators

Every resource has an identity in order to manipulate it. REST services use standard Uniform Resource Locator (URL)[2] semantics to identify the resources being retrieved or modified. The most frequently used, but not complete, elements of a URL are:

scheme://domain/resource_path?query_string

2.1.2.1 Scheme

Since REST uses HyperText Transfer Protocol (HTTP) as its fundamental protocol, the URL scheme will be either "http" or "https", for requests over unsecured and secured connections respectively.

2.1.2.2 Domain

The domain is the Domain Name Service (DNS) name entry that will resolve to a network location, typically a server, which holds the resource. For example, *maps.google.com* is the

[1] "Architectural Styles and the Design of Network-based Software Architectures," DISSERTATION submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Information and Computer Science by Roy Thomas Fielding, 2000.

[2] There is an important, but not significant in this context, distinction between a Uniform Resource Identifier (URI) and a Uniform Resource Locator (URL). A URI is the general class of names, while a URL is a particular type in that it is a URI that resolves to a resource. In this discussion, as in Roy Fielding's dissertation, our focus will be on names that resolve, so we will always use the term URL.

domain name for the Google tile server, and *s3.amazon.com* is the domain name for the Amazon storage services.

2.1.2.3 Resource Path

A resource path uniquely identifies a resource contained within the domain. While not required to be a meaningful name, it is common usage that a resource path looks much like a file path and may identify a set of resources of the same type (like a directory path) or identify a single resource (like a file path). Conventionally, a REST URL is comprised of resource collection type and resource identity pairs:

    http://domain.com/<collection>/<item>/<subgroup>/<item>

To return a collection of a given resource type, the following URL would be used:

    http://domain.com/<collection>

Best practice encourages that the set of resources, represented by a collection of URLs, be of the same resource type. For example, if resources in the collection are expressed via Extensible Markup Language (XML), they should all conform to the same XML schema.

To return a single resource within the collection, the following URL would be used:

    http://domain.com/<collection>/<item>

2.1.2.4 Querystring

Best practices encourages that URL querystring in RESTful services be used to control the view into data and not to be used to provide either "identification" data which should be part of the resource path, or any functional semantics beyond what the HTTP verbs imply. One of the problems of placing identification data in the querystring are that programs can no longer check equivalence by just looking at the URL base and resource path. Consider the following three URLs:

    http://business.com/invoices?invoiceId=10000&lineItem&=3
    http://business.com/invoices?lineItem=3&invoiceId=10000
    http://business.com/invoices/?acceptType=excel&invoiceId=10000&lineItem=3

Intuitively, we recognize that all three reference the third line item of invoice 10000. However, from a machine standpoint, checking the equivalence of these URLs becomes complex and problematic. However, by keeping the identifying information in the resource path, a single representation of the URL emerges and software can do simple identity comparisons with URLs by simply dropping the Querystring:

    http://business.com/invoices/10000/lineItem/3

Another common "anti-practice" is packaging "business operations" into the URL query string. These types of service invocations often package some action, command, or verb into the URL. For example:

http://business.com/invoices&action=addLineItem&partNum=6543&quantity=1&cost=1.00

In general, the querystring should not be used to control the output of the REST operation and directly impact the state of the resource or server. Most often, querystrings are used with REST operations that return collections of resources. For example, the following URL references the set of all invoices belonging to business.com:

http://business.com/invoices

However, a complete referenced collection may be too large to return to the requester in a reasonable time frame or require too much server and network bandwidth to retrieve and transmit (business.com made millions of invoices). Querystring parameters can then be applied as constraints or filters to indicate to the service to only return the subset of resources for which the requester has interest. For example, the following URL only returns the invoices that were invoiced in January 2010 (dates are represented as YYYYMMDD).

http://business.com/invoices?earliestInvoiceDate=20100101&latestInvoiceDate=20100131

Querystring parameters can either control the maximum number of results returned or control a "window" onto a larger collection. This is commonly called pagination in the web community. The following example is a query that returns the next 20 invoices starting at the 80th invoice ordered by invoiceDate:

http://business.com/invoices?numInvoices=20&startIndex=80&sortOrder=invoiceDate

2.1.3 Representation

REST makes an important distinction between the representation of a resource and the resource itself. Clients can manipulate many different forms of the resources stored on servers. A client negotiates with the server to agree upon a representation that the server can produce and that the client can handle using the Internet Media Types registered with the Internet Assigned Numbers Authority (IANA) and documented in RFC 2046.[3] If a server is unable to deliver a resource in a requested representation, it returns an error.

In practice, there are three techniques for requesting the representation type of a resource.

2.1.3.1 HTTP Request Headers

The HTTP protocol specifies the use of the "Accept" request header field for requesting representation types. The "Accept" field may contain a number of content types with explicit preferences that the client will accept. A server is obligated to provide the resource in the representation with the highest client preference or to return an error when it cannot provide the resource in any of the requested content types.

For example, a basic HTTP Request that returns the XML representation of invoice 11111 is:

[3] The name of the original specification was the Multipurpose Internet Mail Extensions (MIME). A full list of the current authorized types is at http://www.iana.org/assignments/media-types/.

```
GET http://business.com/invoices/11111 HTTP/1.1
Accept: text/xml
```

And a basic HTTP Request that returns the JavaScript Object Notation (JSON) representation of invoice 11111 is:

```
GET http://business.com/invoices/11111 HTTP/1.1
Accept: application/json
```

2.1.3.2 Media Type Query Parameter

Most browsers do not readily allow users to modify the acceptable content types of the HTTP Request Headers. To get around this problem, developers of RESTful services have devised mechanisms for embedding information about the desired representation in the URL. Many REST services will accept a special query parameter such as "mimeType". For example:

http://business.com/invoices/11111?mimeType=text%2Fxml
Returns the XML representation of invoice 11111. Note that the slash ('/') character in the query parameter value must be URL-encoded.

http://business.com/invoices/11111?mimeType=application%2Fjson
Returns the JSON representation of invoice 11111

2.1.3.3 Declared Resource Path Extension

An alternative to the query parameter approach uses the file extension of the required representation type at the end of the resource path to identify the requested content type. Ruby on Rails is an example framework that employs this approach.

http://business.com/invoices/11111.xml
Returns the XML representation of invoice 11111

http://business.com/invoices/11111.json
Returns the JSON representation of invoice 11111

2.1.4 Context Free Request and Reply

RESTful services use the strict request and reply pattern of HTTP for invocation. A service consumer (i.e., client) sends a request to a service provider (i.e., server) in the form of the HTTP request. The service provider then responds with an HTTP response.

It is important to note that in the RESTful pattern, the client holds the conversation state in all client-to-server interactions. The employment of context free request and response exchanges is critical to the tremendous scalability enjoyed by the Web today.

3 Design Patterns: The Operations

REST uses HTTP operations to modify resources. An HTTP GET operation retrieves a representation of a resource identified by a URL and never changes the actual resource on the server. An HTTP PUT operation creates or replaces an information resource with a provided representation of it. HTTP DELETE removes a resource from the provider. The POST operation is extremely powerful; it presents information to a resource, typically a form processor that may create, delete, retrieve, or modify resources in complying with the supplied information.

This restricted set of operations makes working within the REST paradigm different from working with other data exchange protocols. Protocols such as Remote Procedure Call (RPC) and Simple Object Access Protocol (SOAP) allow developers to define their own methods as well as the objects passed through these methods. REST only allows the developers to define their own objects. The advantage of this approach is that it is a well understood standard for what is similar to the Create, Retrieve, Update, Delete (CRUD)[4] operations against the information resources.

Software developers will often overload the HTTP operations when implementing their own actions. Since HTTP is perhaps the most widely accepted protocol and most frequently opened port through firewalls, HTTP becomes overloaded with embedded protocols (e.g., SOAP over HTTP) that relegate it to a transport level protocol instead of an application level protocol as intended. While perhaps sometimes required by the engineering need, these implementations risk violating the principles articulated in REST. A REST-compliant exchange should use only HTTP verbs and manipulate representations to work with an information resource.

3.1 HTTP Methods

There are eight methods defined in the HTTP protocol as outlined in RFC 2616. These are GET, PUT, POST, DELETE, HEAD, OPTIONS, TRACE, and CONNECT. Of these methods, REST services are primarily concerned with GET, PUT, POST, and DELETE.

The RFC 2616 specification defines two important concepts when dealing with the HTTP methods: “safe” and “idempotent”. “Safe” means the HTTP operation should have no “significant action” other than retrieval, and be “side-effect” free. In other words “safe” methods should not modify the state of the referenced resources. They should be “safe” for requesters to use without any unintended actions or consequences.

“Idempotent” means that that the results of sending the request multiple times should be the same as sending it once. A common example is the PUT operation which is used to “put” a

[4] Create, Retrieve, Update, and Delete operations familiar to database developers. We should note that the analogy is not complete; the four database operations do not map directly to HTTP verbs.

resource on the “server”. Making the same “PUT” request multiple times for a given resource should leave the resource in the same operational state (audit logs notwithstanding).

The following table shows the HTTP operations commonly used with RESTful services and their basic “semantics” as defined under RFC 2616.

Table 1. HTTP Operations

Method	**Safe**	**Idempotent**	**Description**
GET	Yes	Yes	Used to retrieve a resource referenced by the passed-in URL. The required representation to be returned by the GET can be declared in the “Accept” HTTP Header allowing for multiple representations to be returned for a single resource (i.e., an XML, JSON, HTML, etc.,). A conditional retrieval based on the modified time can also be expressed by using the “If-Modified-Since” header.
PUT	No	Yes	Used to associate the representation content of the request body (typically some XML or JSON payload) to the referenced URL. If the URL does not exist, then a new resource is created using passed representation. If the URL refers to existing resources, then it is replaced by the representation in the request body.
DELETE	No	Yes	Used to remove a resource associated to the passed-in URL.
POST	No	No	Used to submit data such as a form to a server to perform some processing against a resource. May involve the creation of a new resource or appending additional data to an existing resource.

POST Operations

The RFC 2616 does not clearly identify the intentions of the POST method. Hence the exact nature and best practices associated with the “POST” method is a subject of much debate. In general, it is agreed that POST should be used if:

- The service is creating and identifying a new resource (i.e., generates the Id) based on partial information and returning the “completed” resource with Id.
- The service is adding information to an existing resource without passing a complete new state. This includes information such as annotations, additional data items, and modifications (based on partial state).

There is no consensus on when to use POST vs. PUT. The general rule of thumb is, when passing the complete state of a resource to create or replace it, then use PUT. When creating a new resource from partial information, then use POST. Also use POST when modifying an existing resource based on passed-in information or attributes. For example, a POST could increment a total count or change some single text field.

3.2 Representing Collections

As discussed previously, it is possible to have URLs that reference and invoke RESTful services that return collections of resources. To answer these requests, a RESTful service will need to package multiple objects in to a single response. Best practices on recommended ways to package these collections have yet to emerge. However, the three primary methodologies for handling collections with RESTful services include container objects, feeds, and multipart responses.

Container Objects

By far the most common method of packaging collections of resources is to “wrap” them into a higher level “collection” object. For XML representations, this is done by embedding the resources XML into a root collection or list element. This can also be done for collections represented in JSON. If this collection is described in an XML Schema, typically each resource schema type will have a corresponding resource collection schema type, vs. a generic collection type. This enables tools such as XML object-binding code to properly validate and generate the parsing code for these collection objects.

The advantages of this method include being simple and well understood. This method is also easier to implement, especially since it benefits from better tool support, especially from code generators. Perhaps the biggest disadvantage is the introduction of new “collection” types which correspond to each resource type. This makes generic processing of collections and lists a bit difficult. Heterogeneous lists which support more than resource type, especially those types that are not known at development time, can also become a problem.

Feeds

Feeds are commonly seen on the Internet as Atom or Real Simple Syndication (RSS) news feeds. These news feeds typically provide a list of hyperlinks with corresponding summary information to content items or resources on a site. The goal of a news feed is to allow the consumer to review large collections of summary items without having to pull the entire content for each item. These feeds also support a polling model in which an application known as a feed reader or aggregator could periodically poll the feed to determine if there are any updates to the content associated to the feed.

As stated previously with RESTful services, it is usually beneficial to reuse existing representation standards and models. Since feeds are basically a container object for generic content, the existing feed standards can be reused to return resource collections. The two primary standards for feed representations are RSS and Atom. RSS is able to represent a list as a collection of annotated hyperlinks to the resources. Like RSS, Atom also provides a list of annotated hyperlinks, but also allows embedding the actual content in an entry's "content" tag. This prevents the requester from having to send a separate request for each feed entry to retrieve the content associated with that entry.

The advantage of using feeds to represent resource collections are that the RESTful services will behave like a web news feed. This enables the existing feed readers and aggregators to monitor or poll the content of this RESTful service just like a normal news feed. The RESTful service can then support both human and machine consumers, using existing applications. It is important that such services implement HTTP "If Modified Since" header behavior to prevent returning duplicate responses to the periodic queries that occur when aggregators poll the feed.

A disadvantage of this method is that the feed type is a generic XML collection that does not specifically identify the types of resources it contains. This can be problematic with XML binding and code generation tools that developers use to alleviate the time-consuming work of parsing XML. There are potential ways around this by using sophisticated XML schema extension techniques.

Multipart HTTP Responses

The HTTP protocol enables the passing of multiple resources in a single response message body by the use of Multipurpose Internet Mail Extensions (MIME) multipart content types (defined in RFC 2046 Section 5.1). This enables a collection of resources to be passed to a requester in a single HTTP response without the use of any extra "container" or feed semantics. This method is not commonly used or known, but is gaining popularity with the advent of "HTTP Push" or "Comet" techniques.

Comet techniques use long-running multipart responses across persistent HTTP connections to enable servers to "push" content to the browser. The advantage of using this technique with RESTful services is that long-running queries, or queries with large result sets, can start trickling resources back to the requester before the query has been completed. This enables the requester to start utilizing these resources immediately without having to wait for the full set of results. This alleviates a huge disadvantage of XML, in that to easily construct and return an XML container element or feed, the complete set of returned resources have to be known. The disadvantage also manifests on the receiving end, in that many XML parsers and technologies require the complete document, before allowing usage. Constructing an XML document from a large collection can also consume tremendous amounts of server resources, especially since these results can be held in server memory until transmitted to the requester. This tremendous performance and resource advantage is starting to drive more developers to look at his methodology despite the technical difficulties in leveraging it.

4 Documenting RESTful Services

Like any interface or Application Program Interface (API), RESTful services need some level of documentation to be useful to an external consumer or developer. REST does try to keep documentation light by reusing the existing specifications and standards which include primarily the HTTP specification. By leveraging the HTTP specification, the details of constructing valid requests and responses, exchanging them over a network, describing the set of allowed operations (i.e., GET, POST, PUT), and even generating errors have already been worked out and do not need additional documentation. This leaves five primary pieces of documentation a RESTful service developer needs to provide to their consumers.

1. Describe the resource
2. Describe representations
3. Describe the details on how to form a valid request in the form of a URL
4. Detail any actions or behaviors performed by a RESTful service request beyond the HTTP basic operations
5. Set the terms of usage for the service

RESTful services are primarily described and documented with human-readable text. This is a departure from SOAP-style services which heavily leverage machine-readable specifications known as Web Services Description Language (WSDL) definitions. The advantage of a machine-readable specification is that it allows software tooling to generate scaffolding code that assists in invoking and providing a service. It also allows for automated validation of exchanged information and content. One of the primary reasons the REST community did not adopt a machine-readable specification is that the existing standards, such as WSDL or Interface Definition Language (IDL), were incapable of describing a RESTful interface. Another reason is that the existing specification standards were considered too complex to describe simple RESTful exchanges. With the advent of new standards such as WSDL 2.0 and the less complex Web Application Description Language (WADL), RESTful services can now be described in machine-readable documents. This enables the RESTful services to have the same tooling support and advantages originally only enjoyed by SOAP and RPC-style interfaces.

Describing SOAP vs. REST Operations

A SOAP-service endpoint has a clearly defined boundary with a well defined set of operations. The set of operations supported by a SOAP service are clearly laid out and documented with a WSDL definition. The SOAP-service endpoint even has its own identity with the service URL, which provides the location for invoking the service and for accessing its definition.

A RESTful service on the other hand is not so clearly defined. A single RESTful service may be defined as an arbitrary collection of HTTP operations working against an arbitrary set of resources. Perhaps this grouping was established because they share a common underlying code base, infrastructure, or development team. Many times a single RESTful service describes a set HTTP operation against a single type of resource. REST does not have a clear URL that "points" to clearly documented, well defined, machine describable set of operations that are included into a well-defined service endpoint as described by SOAP-based services. At the end of the day, a RESTful service endpoint boils down to a grouped set of operations (commonly called methods) that work against a set of resources. The "how" and "why" these methods are grouped into a single RESTful endpoint is largely irrelevant to the invoker, since this information is not needed to make a RESTful service work (unlike a SOAP endpoint in which the invoker needs the service endpoint URL before invoking any operation).

4.1 Describing the Resources

It is always helpful to have a general description of the actual resource being exchanged. The general description should include a brief statement or definition that describes the resource being exchanged. This establishes context which enables the engineer to verify that the resource type is indeed what they required and are expecting. This is especially important if the service is providing information using heavily overloaded terms. The classic example is a service that returns information for a "tank". It can be difficult to discern whether it is a storage tank, a tracked vehicle with big guns tank, or a video game character archetype tank. Often, the knowledge of the source itself will provide the appropriate context (i.e., http://homedepot.com it will provide info on storage tanks, http://janes.com will provide info on tracked vehicles with big guns, http://worldofwarcraft.com will provide info on the later). However, it is a mistake to assume the consumer will be able to discern the type of resource directly from knowledge of the source. (What kind of tank information will http://www.armysurpluswarehouse.com provide?)

For structural information, in which the content is broken up into some arrangement of data items or fields, a description of each item should be provided. The description should be fairly generic and independent of the actual format or representation that gets transmitted. This is because a RESTful service may be able to return multiple representations of a resource, such as XML, JSON, or Excel. These representations will usually include the same information and data items regarding the resource, only packaged in different formats.

4.2 Describing the Representations

A resource is a concept; a representation is the actual characters or binary bits that convey a resource between consumers and providers. A RESTful service may be capable of producing several different kinds of representations of a resource on request. Documentation should list all supported representations by their content type.

The contrast between SOAP handling of information and the REST paradigm is significant. SOAP typically works only with XML documents and attempts to encode other types of information within the XML document. The documentation for these services must describe the nature and arrangement of the component data elements and clearly spell out methods for interacting with the information hidden behind the service and provide machine-readable descriptions known as schemas. Schemas, and similar techniques for describing the requests, responses, and sequence of flow, allow consumers to use automated software generation tools to develop client applications.

Under REST, representations enjoy a rich variety of more than 350 content types that also include the SOAP document type. A RESTful service may provide images and video or may exchange invoices, medical records, or contact information. Information exchanges typically use well established text or binary formats that are documented elsewhere. For example, Joint Photographic Experts Group (JPEG) and Portable Network Graphics (PNG) are common representations for images, Moving Pictures Experts Group (MPEG) is a common representation for video data, and Windows Wave (WAV) is a common format for sound recordings. For these representations of the underlying resource, simply declaring supported exchange formats is sufficient. Usually this is done by listing the Internet Media Types associated with the resource.[5]

Structured textual representations need to be documented to be useful. Structured information using individually crafted formats employing XML or JSON require documentation because their encoding, by its nature, cannot convey meaning. The provision of details on the structure of information also allows consumers to check for correctness. XML-based representations use schemas in the same fashion as the SOAP protocol; JSON-based representations typically use documentation intended for people to read. Machine readable specifications exist for JSON-based representations, but they are not in wide use.

Ideally, for a RESTful service, the information model and the access method are the same; one does not have to understand the information and then learn how to access it. The degree of success in achieving this goal will depend on the laying out of the structure of the information and then constructing resources that provide it. The software implementation of the service then handles only the details of producing the information and converting it into a requested content type.

4.2.1 Constructing the URL

The documentation for a RESTful service needs to describe how to construct a URL to reference either a single resource or collection of resources. Generally, a URL consists of a base, the resource path, and the acceptable parameters that can be included in a query string.

Base

Typically, the base of the URL identifies a collection of resources that have the same type or some common relationship that allows grouping together. The following, previously used example, demonstrates a URL to a collection of invoices for business.com:

[5] This information can, of course, be obtained at run time through the use of the HEAD verb to request the so-called metadata about a resource; but in practice, this technique is not used.

http://business.com/invoices

Path

A path denotes a way to reach a specific resource, often as a series of progressively finer descriptions or greater detail. The following URL identifies a single line item within an invoice by walking down the hierarchy from the unique identifier through the line number.

Query String

Query strings often filter particular information from a set of data or qualify the representation returned. A specification must include all optional and required parameters along with acceptable values and legitimate combinations:

http://business.com/invoices/<InvoiceId>/line/<LineNumber>?lang=en-us&maxlines=25

4.2.2 Describe the Operations

When creating a resource, one has the option of specifying which operations it supports. Because REST relies on the HTTP protocol, the operations should align with the defined semantics of the HTTP verb. For example, the HTTP GET operation should "retrieve" or "read" something. A PUT should "create" or "replace" a resource. DELETE should "remove" it. Finally, POST operations should interact with a resource, either "modifying" or "manufacturing" it directly or initiating other activity through the form processer.

4.2.3 Describe the HTTP Headers

The most common HTTP headers that RESTful services include are "Accept", "Content-Type", and "If-Modified-Since". The "Accept" header declares the content type(s) acceptable to the invoker. The "Content-Type" header declares the content type (i.e., mime-type) of the body of the request for HTTP PUT and POST requests. This is useful if a RESTful service allows different representations to be uploaded, such as JSON and XML. The "If-Modified-Since" header requests that the service return only a "304-Not Modified" response if the content is unchanged after a certain date.

4.3 Example

The following table is an example of the documentation for a basic invoice RESTful service provided by "business.com". This invoice service reuses the OASIS Universal Business Language (UBL) XML representation for invoices. The RESTful service enables the retrieval and modification of existing invoices, and enables the creation of new invoices.

Table 2. Business.com Invoice REST Service

Business.com Invoice REST Service

The business.com invoice service enables the storage and retrieval of the invoices that are currently or have been processed by business.com. Invoices are exchanged using an OASIS Universal Business Language (UBL) 2.0 XML as defined by the UBL XML Invoice Schema.

This service has five Methods which provide the ability to retrieve a set of invoices, retrieve a single invoice, create a new invoice, update an existing invoice, and delete an invoice.

Terms of Usage

Retrieve Invoices

Description:

This operation will retrieve a set of invoices based on the set of criteria passed in the query string.

Base URL:

`http://business.com/invoices/`

HTTP Operation:

GET

QueryString Parameters

The following parameters can be passed to the service to control the set of invoices returned in the collections.

earliestIssueDate	Excludes all invoices that were issued before the passed-in dates from the returned result. Dates are based on GMT time zone and are encoded as YYYYMMDD.
latestIssueDate	Excludes all the invoices issued after the passed-in date.

customerCountries	Includes only invoices that were sold to customers within the passed-in set of countries. Countries are passed using two-character country codes as defined by ISO3166. Multiple countries are comma delimited (i.e., U.S., UK, FR) and maximum of 10 countries can be passed.
lowAmount	Excludes all invoices in which the total sales amount was less than the passed-value in U.S. dollars.
highAmount	Excludes all invoices in which the total sales amount was more than the passed-value in U.S. dollars.
state	Can either be of the values of SUBMITTED, PROCESSED, SHIPPED, RECEIVED.

HTTP Headers Supported

If-Modified-Since	If this HTTP header is set, then the service will return a “304-Not Modified” HTTP result if no invoices were added, modified, or removed since the passed-in date. This supports the common “feed” semantics of only returning a feed if it has changed.

Response

Invoices will be returned in an Atom feed in which each entry’s “content” element contains a complete UBL 2.0 “Invoice” element as defined by the UBL Invoice Schema. The feed will be sorted from most recent to least recent based on the “Issue Date” of the invoice. A maximum of 100 Invoices will be returned. If no invoices match the passed-in filter, an HTTP 404-Not Found will be returned.

HTTP Response Codes

200-OK	Success
404-Not Found	Returned if no invoices were found to match the passed-in criteria.
403-Not Modified	Returned if the “If-Modified-Since” header is set and no invoice has been added, updated, or deleted since the passed-in date.

Examples:

```
http://business.com/invoices/?customerCountries=%22US,UK%22
```

Returns the invoices for customers in the U.S. and UK. http://business.com/invoices/?earliestIssueDate=20100101&latestIssueDate=20100131 Returns all invoices for the month of January 2010.

4.4 Machine-Readable Specifications

SOAP-based services are heavily reliant on machine-readable specifications expressed in WSDL. WSDL is a specification for how to define and declare a web-service endpoint interface in an XML document, simply called a WSDL file or simply the "the WSDL". The WSDL primarily details the operations and XML messages utilized by these messages. The advantages of such a machine-consumable description include the following:

- Automated validation of web-service messages. Validation software could check incoming messages for correctness and "well-formedness" before passing it to the web service for processing.
- Automated code generation and scaffolding. Tooling enabled developers to quickly generate the required code to both consume and provide web services with WSDL descriptions.
- Useful for brokering machine interfaces between multiple organizations and communities. Standards bodies have determined that WSDLs and XML schemas are especially useful for expressing standard interfaces and models.
- Useful for endpoint discovery and dynamic resolution of service endpoints.

Early REST proponents argued against the utilization of these machine-readable specifications and opted for just human-readable documentation. Many of their reasons include the following:

- WSDL 1.x was unable to describe the HTTP semantics required by REST interactions. The standards and technologies that were originally available were engineered for SOAP interactions, not the simplified REST models.
- It was argued that SOAP, WSDL, and related technologies standards were too complex and too heavy weight for most of the basic interactions that were happening on the web.
- WSDLs were often used as a substitute for proper human-readable documentation. Since RESTful services could only be documented via human-readable text, it could

be argued that they produced superior documentation which made them easier to use (despite the superior tooling available for SOAP).

- The advent of Rich Internet Applications (RIA) and corresponding technologies, such as Asynchronous Javascript and XML (AJAX) and JSON, were much better suited for RESTful-style services over SOAP-based services. RIAs largely eliminated the tooling advantages of SOAP since the tooling was designed for "heavy clients". JSON, a browser friendly, non-XML data representation became widely used for RIAs due to ease of use and performance advantages within the browser. Wide adoption of JSON representations, which were incompatible with SOAP and WSDL, also drove developers away from both SOAP and WSDL's machine-readable descriptions.
- Code generation and automated validation made interfaces brittle. It is often argued that generated clients and stubs were less forgiving to small interface changes and nuances than human-coded invocations, which again mitigates the value of machine-readable descriptions.
- Original SOAP services and their corresponding WSDL definitions interfaces were often of poor quality since they were largely machine generated from legacy code using SOAP's advanced tooling. RESTful interfaces were largely "hand-crafted" due to the lack of tooling, and so were seen as having superior quality.
- Finally the web-service marketplace never emerged. Vendors and providers resisted complying with standards to deliver "low-profit" commodity services. They instead delivered highly proprietary services which enabled them to deliver differentiated value and to better lock-in customers. This is perhaps why the storage services offered by companies such as Amazon (Simple Storage Service), Microsoft (Azure), and Google (Google Docs) have very different APIs (although all are using REST).

Despite the disadvantages, many developers have missed the advantages of machine descriptions such as automated code generation and scaffolding, automated validation, and a robust interface specification. To this end, many potential description languages have emerged as the various communities attempt to converge onto a small set of description languages standards that support RESTful services. Currently, the two leading candidates that have the largest adoption and the most tool support are WSDL 2.0 and WADL.

It is important to note that REST machine-readable specifications and description languages are not in wide use. Many REST proponents argue that machine descriptions are still overly complex and largely unnecessary. They also argue that code generated from these descriptions can be "brittle" and is prone to break on small interface changes.

4.4.1 Web Services Description Language 2.0

WSDL is an XML grammar for describing protocols, message formats, and operations needed to support interactions between one or more systems that typically communicate on a TCP/Internet Protocol (IP) based network (i.e., the web). WSDLs are often called network-based APIs, which allow intersystem communications (vs. the standard software code API which is typically used for intra-system communications). WSDL is very general purpose and can be used to describe a wide range of interfaces using various data representations and network protocols. However, despite this flexibility, an interface described by a WSDL definition is almost always based on an XML data representation transmitted using SOAP over HTTP (this is the "classic" definition of a web service).

In response to the large developer movement toward RESTful interfaces and to reduce WSDL 1.x's complexity, the W3C Web Services Description Working Group released the WSDL 2.0 Recommendation in 2007. WSDL 2.0 provided a simplified definition of web services and introduced a much enhanced HTTP binding specification aligned to describing REST invocations. This specification enabled developers to map XML types and elements to URL context paths and query string. This enabled the specification of dynamic and parameterized URLs critical in the description of RESTful services. The HTTP Binding specification also enabled the declaring of the supported HTTP operations and header variables which can also be used by a RESTful service.

The following XML samples show a WSDL 2.0 example of the Yahoo Web Search REST Service. This service takes a set of keywords, a maximum number of results, and a start index. It returns a set of search results from the Yahoo search engine. The first XML sample shows the XML schema for the "Result Set" resource which is used to represent the search results returned by the service. The second XML sample is the actual WSDL 2.0 interface specification of the REST service.

Please note that for brevity the original WSDL was edited to scale it down for this paper.

XML Schema for WebSearchResponse.xsd

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="urn:yahoo:srch"
targetNamespace="urn:yahoo:srch" elementFormDefault="qualified">
  <xs:element name="ResultSet">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Result" type="ResultType" minOccurs="0" maxOccurs="100"/>
      </xs:sequence>
      <xs:attribute name="totalResultsAvailable" type="xs:integer"/>
      <xs:attribute name="totalResultsReturned" type="xs:integer"/>
      <xs:attribute name="firstResultPosition" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="ResultType">
    <xs:sequence>
      <xs:element name="Title" type="xs:string"/>
      <xs:element name="Summary" type="xs:string"/>
      <xs:element name="Url" type="xs:string"/>
      <xs:element name="ClickUrl" type="xs:string"/>
      <xs:element name="ModificationDate" type="xs:string" minOccurs="0"/>
      <xs:element name="MimeType" type="xs:string" minOccurs="0"/>
      <xs:element name="Cache" type="CacheType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="CacheType">
    <xs:sequence>
      <xs:element name="Url" type="xs:string"/>
      <xs:element name="Size" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

WSDL 2.0 Definition of the Yahoo Web Search REST Service

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:description xmlns:wsdl=http://www.w3.org/ns/wsdl
  xmlns:yahoosrch="urn:yahoo:srch" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:whttp="http://www.w3.org/ns/wsdl/http"
  xmlns:wsdlx="http://www.w3.org/ns/wsdl-extensions"
  xmlns:ysearchtypes="http://wso2.org/ns/2007/yahoo/srch/types/"
  xmlns:tns="http://wso2.org/ns/2007/yahoo/srch/"
  targetNamespace="http://wso2.org/ns/2007/yahoo/srch/">
 <wsdl:documentation>
  WSDL 2.0 description for Yahoo REST Search API.
 </wsdl:documentation>
 <wsdl:types>
```

This imports the WebSearchResponse schema defined above.

```
   <xs:import
    namespace="urn:yahoo:srch"
    schemaLocation=" WebSearchResponse.xsd"/>

   <xs:schema targetNamespace="http://wso2.org/ns/2007/yahoo/srch/types/"
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:tns="http://wso2.org/ns/2007/yahoo/srch/types/"
     elementFormDefault="qualified" attributeFormDefault="unqualified">
```

Note that the SearchString element's child elements are mapped to URL Querystring parameters by the HTTP Binding.

```
      <xs:element name="SearchString">
        <xs:complexType name="SearchString">
          <xs:sequence>
            <xs:element name="query" type="xs:string"/>
            <xs:element name="results" type="xs:integer"/>
            <xs:element name="start" type="xs:integer"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </wsdl:types>
  <wsdl:interface name="Search">
    <wsdl:operation name="YahooSearch"
     pattern="http://www.w3.org/ns/wsdl/in-out"
     wsdlx:safe="true">
      <wsdl:input element="ysearchtypes:SearchString"/>
      <wsdl:output element="yahoosrch:ResultSet"/>
    </wsdl:operation>
  </wsdl:interface>
```

This is the HTTP Binding which maps the Search interface to an HTTP GET operation.

```
  <wsdl:binding name="YahooHTTPBinding" interface="tns:Search"
   type="http://www.w3.org/ns/wsdl/http" whttp:version="1.1">
    <wsdl:operation ref="tns:YahooSearch" whttp:method="GET"
     http:inputSerialization="application/x-www-form-urlencoded"/>
  </wsdl:binding>
```

The service element declares a RESTful service endpoint with the base URL at the http://search.yahooapis.com/WebSearchService/V1/webSearch

```
  <wsdl:service name="YahooSearch" interface="tns:Search">
    <wsdl:endpoint name="YahooREST" binding="tns:YahooHTTPBinding"
     address="http://search.yahooapis.com/WebSearchService/V1/webSearch"/>
  </wsdl:service>
</wsdl:description>
```

4.4.2 Web Application Description Language

WADL was proposed to the W3C as a submission by Marc Hadley at Sun Microsystems. Unlike WSDL, which is a generalized description language for describing service interfaces, WADL was specifically designed to describe RESTful services, which by definition, always uses the HTTP protocol. By specifically addressing REST, WADL removes some of the ambiguities that appear in WSDL and is argued to be a much more understandable and succinct description of RESTful services vs. WSDL 2.0. Appendix B provides a WADL description of the same Yahoo Web Search interface shown previously.

This Page Intentionally Blank

5 Service Management

Throughout its lifecycle, a service will inevitably evolve and undergo changes. To minimize impacts and potential breakages with consumers, this change must be carefully managed for both SOAP and RESTful services.

5.1 Simple Network Management Protocol

Application Management should conform to the Simple Network Management Protocol (SNMP) Version 3.

5.2 Service Heartbeat

A compliant service should support Heartbeats as outlined in Section 7.2.5 of the Web Services Management (WS-Management) Specification. Reference: "Web Services for Management (WS-Management)," April 2006.

5.3 Versioning

SOAP-based services typically manage change by providing a new endpoint each time the interface changes. SOAP enables this behavior by having clear, well-bounded service interface definitions (i.e., WSDLs) and endpoint "identities" (service URLs). What this means is that each time a SOAP interface changes, it gets delivered as a new service endpoint with its own unique UR so service consumers can select the "version" of service they require by simply accessing the endpoint URL. This enables multiple versions of a service to coexist, and allows consumers to migrate to newer versions at their own pace.

A RESTful service, on the other hand, is not so clearly defined. REST URLs refer directly to resources, not service endpoints. What this implies is that a RESTful service does not support clear endpoint URLs identifying one revision from another. Also the exact boundary of a RESTful service is often ambiguous to the consumer and may only be known to the developers. The consumer is left with a set of URLs that reference known resources, with little or no revision or version information. These URLs are also long lived as they get used as hyperlinks in external documents or other systems. The exact revision of a RESTful service is largely hidden from the consumer, whereas for SOAP services, the consumer is directly aware of the particular revision they are utilizing.

To safely evolve a RESTful service, the developer must consider what is safe to change with minimal disruption for their consumers. This is especially troublesome for large scale RESTful service providers such as Amazon, Google, etc., in which a small change can lead to tremendous, potentially disastrous, disruptions to a very large population. The Internet itself is heavily reliant on these interfaces to be stable. To this end, changes in these changes to these services must be assessed to determine potential impacts to the end consumers.

Low risk changes that have minimal chance of impacting end consumers are considered "safe". Examples of low risk changes include the following:

- *Additional query parameters for filtering collections*.
- *New additional representations*. These representations can be selected by the usual "accept type" methods discussed previously.
- *Additional HTTP Operations supported*. In general, if a read-only RESTful service that only supports "GET", expands to support "PUT", there should be no impact on existing consumers.

Medium risk changes have a high chance of impacting at least some small group of consumers. Examples of medium risk changes include the following:

- *An existing representation change*. This occurs if the default representation (if no accept type is declared by the consumer) or if the representation of a known accept type has changed. XML has methodologies for augmenting existing type, such as namespaces, to retain backward compatibility. However there is no guarantee that existing consumers will properly handle these changes.
- *HTTP Operations change behavior*. A developer can potentially break consumers by changing the expected behavior of existing HTTP Operations. The HTTP POST operation is especially vulnerable to shifts in behavior due to its ambiguous definition.
- *Changing Query Parameters*. Removing or changing the behavior of a query parameter, such as used in collection filters, may cause difficulties to a subset of consumers.

High risk changes have broad impact over a potentially large number of users. Examples of high risk changes include the following:

- *URL Path Changes*. Changing how the URL is structured is almost always going to cause some broken links somewhere.
- *Deprecated representations*. Removing an existing, used representation will also cause some consumer problems.

Versioning RESTful Services

In general, RESTful services are typically not versioned in a meaningful way to the consumers (it is assumed the developers have some versioning on the code base). It is rare to find version numbers in any RESTful interface and its corresponding documentation. This is because RESTful service providers usually provide only one version of the interface, the "current version". This is why change management and backward compatibility are critical to design and evolution of these services. It also means developers must be very careful with the initial design of the RESTful interface, since once it is publically available and successful, it is very difficult to fix interface issues without massive consumer disruptions. Indeed, it is the stable and immutable nature of RESTful interfaces that have helped propel and scale up their usage across the Internet. The existence of the World Wide Web itself relies on very stable interfaces and URLs to facilitate the hyper-linking across web resources.

Despite the lack of a public versioning scheme for RESTful services, versioning does factor heavily into the representations that a RESTful service can return. RESTful services are fully capable of returning multiple representations of a resource using multiple versions of some data standard (i.e., for example HTML v3.0, 4.0, 5.0). The service needs to clearly indicate which representation data standard versions are used and how the consumer asks for specific versions.

5.4 Service Identification

Services are uniquely identified based on the "Standard for Naming Active Entities on DoD IT Networks." The identifier is formatted as a DoD Public Key Infrastructure (PKI) Subject Distinguished Name and an example of the format is:

CN=4 to 128 character service name provided by the service's owning organization

serialNumber=<UUID>

OU=<name of COCOM, Service, Agency, or DoD Affiliate>

OU=SERVICE

OU=DoD

O=U.S. Government

C=US

For the complete specification, refer to "Standard for Naming Active Entities on DoD IT Networks", Version 3.1, 1 December 2009 (in the DISR Registry).

This Page Intentionally Blank

6 Discovery

Discovery capabilities enable users to search the Internet for specific pieces of information and content. There are two types of searches that commonly occur in regard to web services. The first is to search for a web service that meets some potential need (a.k.a. Service Discovery). The second is to search within a web service or set of services for some specified needed information, data, or content (a.k.a. Content Discovery).

Service Discovery

Service Discovery involves searching for a web-based capability or service that provides some needed operation, function, or information. After a candidate service has been identified, the consumer will need to interface with the service to utilize it. This requires the service provider to describe and detail how to leverage and utilize the provided service. Often these details are of a technical nature and may require custom software to be developed to interface with the service. Because of this, the primary consumer of service discovery is often the software developer wanting to integrate some existing functionality or service into new capability during its development.

Content Discovery

Content Discovery involves searching for relevant information provided through a service. This type of discovery is performed after a service has been deployed, and is in use. The primary consumer in this case, is the non-technical end user who is only interested in the provided information, not the providing capability or service.

6.1 REST Discovery

Unlike SOAP-based services, REST does not prescribe a specific technology or standard for either service or content discovery. REST relies on external discovery providers such as Google or Yahoo to provide both service and content discovery.

RESTful Service Discovery

To enable a potential consumer to discover provided RESTful services (or any type of web service for that matter), a service provider simply publishes a human-readable description of the service on the web. The service provider can then ask the popular search engines (i.e., Google) to incorporate this description into their indexes, just like any other piece of web content. A consumer then discovers the service by simply performing a standard keyword web search.

RESTful Content Discovery

RESTful content discovery works much like RESTful service discovery, except that instead of submitting the service description to the search engine, the service provider directly submits the URLs of the resources. Because the resources provided by RESTful services appear exactly like web content, the same search engines can be used to index the contents of a RESTful service and enable standard web searches to be performed against it. However there are a few points and rules to consider when using this method of discovery:

- The information provided by the service must support a text-based representation. Images, audio, video, and similar data do not work well with standard, automatic keyword indexing technologies.
- The resource should have enough semantic content to enable itself to be discoverable. For example a RESTful service that returns a single-number temperature for a city (encoded in the URL, such as http://temperature.com/city/Boston/currentTemp) does not provide enough content to be properly indexed.
- The resource may need to attach metadata to provide enough semantic content to be indexed. If the previous example returned "The current temperature in Boston, Massachusetts is 30", it becomes a viable candidate to incorporate into an index.
- The resource should have an HTML based representation that is easily readable in the browser.
- To facilitate indexing multiple resources, the service provider can provide a "page" with links to new or updated resources.

For content that is not easy to index automatically—such as images, audio, and video—several approaches exist. Some of these include the following:

- Metadata Catalogs – Metadata catalogs allow external information to be captured for the resources. The DoD has mandated the DoD Discovery Metadata Specification (DDMS) for discovery metadata, and the use of metadata catalogs for storing DDMS records.
- Tagging – Tagging enables the provider of information to associate a set of keywords or "tags" to content.
- Folksonomies or Collaborative Tagging – Folksonomies allow multiple users to attach their own tags to content. Content that gets associated to the same tag by multiple users gets ranked higher when that tag gets queried.

7 Service Security

Security for RESTful services is confronted with an architectural problem: in any RESTful architecture, the server will not store any client context between requests. In this sense, there cannot be any concept of a client session which makes multi-step authentication and authorization protocols cumbersome.

DoD and IC have published a comprehensive Service Oriented Architecture (SOA) Security Reference Architecture that is the underlying foundation for the Net-Centric Enterprise Services (NCES). This SOAP service-based architecture includes a complex set of authorization models that allow services to fine tune their requirements to their needs. On the downside, the use of SOAP, WS-Security, Security Assertions Markup Language (SAML), and other protocols in this area are very processing intensive and scale poorly to large, complex systems. These overall issues and the general architectural style speak strongly against current attempts to "port" the WS-* specification to the RESTful world, e.g., through efforts such as REST-*.

7.1 General RESTful Security

The following recommendations are for services that are not for DoD-wide consumption, but need authenticated and authorized access:

1. **Transport-Level Security**: In most use cases, transport security is sufficient and no message-level security is needed since there is no intermediate. This is directly in line with the general point-to-point approach of the RESTful architectural style. Transport Layer Security (TLS)/Secure Sockets Layer (SSL) is sufficient for providing mutual authentication, confidentiality, and integrity. Using the DoD PKI, this approach is easy to support and known to scale to large systems. The DoD regards services, including RESTful services, as Non-Person Entities (NPE) that require unique identification. The DoD PKI will be employing an NPE Registry to create Unique User Identifiers (UUIDs) for services. These UUIDs will be used during authentication exchanges for accessing a service. See Section 5.2, "Service Identification."
2. **Standards-Based Authorization**: The protocols for RESTful authorizations are still evolving, but the Internet Engineering Task Force (IETF) has already standardized the Oauth protocol. There are a couple of interesting exchange schemes that are building on top of Oauth, such as the Web Resource Access Protocol (WRAP) or the Kantara User Managed Access (UMA). In addition to various open source libraries used by Internet Service Providers (ISPs) such as Google, AOL, and Twitter, there is emerging vendor support (e.g., Sun OpenSSO). Some of these protocols (such as UMA) can be mapped more or less directly to the SOA Security Reference Architecture by identifying the Policy Decision Point (PDP), Policy Enforcement Point (PEP), etc., within this architecture.

More complex service implementation problems may be addressed as follows:

1. **Service Chaining**: Like SOAP-based services, RESTful services are expected to be orchestratable. For this, the identity of the invoking service, the invocation path, and the end-user should be available to the service provider, if required. There will be many cases, however, where only a subset of this information can be made available.
2. **Secure/Multipurpose Internet Mail Extension (S/MIME) Media Type Encoding**: Message-level security can be implemented by using S/MIME encoding, as described in Section 3 of RFC 3851.
3. **Cross Domain Systems (CDS)**: CDS implementing a RESTful architectural style face issues similar to those of SOAP-based web services. In addition, the payload of RESTful services is not guaranteed to be XML, so additional measures for non-XML media type will need to be implemented.

The remainder of this section describes the technology components that are currently available to create secure RESTful services

7.2 Data Transport Confidentiality and Integrity

For the purpose of this paper, we assume that RESTful services are implemented through HTTP protocol exchanges. As such, SSL/TLS should be used to protect the transport channel between two actors as long as the service has been issued a server certificate. To avoid a recent security issue, SSL session parameter renegotiation shall not be used.

This approach does not address message-level security, and, as such, only allows point-to-point protection and not end-to-end channel protection. For most practical applications this is sufficient at this time. Applications that are interested in high-level cryptographic protection schemes should use S/MIME as content type for protecting arbitrary media types by following the S/MIME construction rules in Section 3 of RFC 3851. This results in payloads of media type "multipart/signed" and "application/pkcs7-mime". A detailed description of this approach is outside the scope of this paper.

7.3 Authentication

In line with DoD guidelines, authentication should be performed through the DoD PKI whenever possible. For an end-user facing applications away from the tactical edge, this presents no major problem since most active personnel are issued a DoD Common Access Card (CAC) with X.509v3 compliant certificates. When accessing RESTful services from a standard browser (e.g., through the use of AJAX application accessing RESTful XML services), users can be authenticated directly through their CAC certificates by requiring HTTP client authentication through TLS.

The NCES Reliable Certificate Validation Service (RCVS) provides an enterprise-wide certificate validation service suitable for verifying the certificates.

For all other situations the following approaches are available:

7.3.1 End-User Without DoD PKI Certificate

Situation: This situation is common in coalition or Government interoperability scenarios, as well as when providing services to the tactical edge. Also, while the DISA and the AF PKI SPO are still in the process of concluding the SIPRNet hardware token trial, the content of this section applies to classified networks as well.

Solution: Servers hosting web services shall use server-side certificates for server authentication through SSL/TLS. Session renegotiation shall not be used. User shall authenticate through HTTP Basic Authentication, using centralized user account stores (such as Lightweight Directory Access Protocol [LDAP]) where possible. Usernames and passwords shall not be cached on the client-side in persistent caches.

7.3.2 Machine Clients With PKI Certificates

Situation: This situation applies to machine clients that have been issued client certificates. While this is not yet deployed DoD-wide, programs may elect to provide machine clients with local certificates as interim solution. Once device certificates become available to all services, this approach will become the preferred way to authenticate machine clients.

Solution: Machine clients that have been issued X.509v3 certificates can authenticate to servers through TLS similarly to end-users. The device client certificate is typically stored in a secured portion of the operating environment and only accessible to the client itself.

Note: At this time, this approach only authenticates the identity of the device. The identity of the end-user or even the entire invocation chain cannot be resolved through this approach. This is a known limitation.

7.3.3 Machine Clients Without Certificates

Situation: When there is no device certificate available to the machine client, there are only a number of relatively insecure means to authenticate the identity of the caller. If possible, this situation should be avoided.

Solution: The machine client may authenticate with a static username/password token through HTTP Basic Authentication. It is critical to manage such passwords in a consistent way and ensure frequent updates to both the server and client. Additionally, the identity of the machine client device may be authenticated through the IP address. Note that this is very insecure without IPSec protection of the underlying IP traffic.

7.4 Authorization

While RESTful web services would be able to utilize a rich ABAC architecture using SAML and Extensible Access Control Markup Language (XACML), these approaches are in many cases too heavyweight and not feasible for a lightweight architecture. The following approach may be used to create an authorization-architecture for lightweight services:

7.4.1 Resource Pattern Matching

Situation: Since RESTful web services operate on resources typically represented by HTTP URLs, authorization to access a given resource may be achieved by specifying URL pattern rules for users and groups of users.

Solution: Most modern web-access server products provide small agents for common application and web servers that act as policy agents on the container. These agents protect resources on the server through a variety of rules and policies. Some products allow policy decision requests to be sent to a centralized PDP, either through proprietary protocols or XACML protocol exchanges.

7.4.2 OAuth Authorization

Note: OAuth, OAuth WRAP, and Kantara UMA are emerging protocols that have been created for authorizing access to RESTful resources. They are built around HTTP protocol and make heavy use of headers to perform authorization.

Basic OAuth was created to facilitate user-authorized resource sharing between two web resources. While this may be applicable to some situations, basic OAuth does not allow the creation of a server- or enterprise-managed authorization model. Kantara UMA introduces an Authorization Manager into the OAuth protocol flow, thus allowing more distributed as well as centralized policy decision.

The entire OAuth protocol suite is currently still sedimenting, and there is only very limited commercial support. At the same time, major vendors have started implementing OAuth and are in the process of integrating this protocol into their web access products.

Appendix A

Glossary

AJAX	Asynchronous JavaScript and XML
API	Application Programs Interface
CRUD	Create, Retrieve, Update, Delete
DNS	Domain Name Service
DoD	Department of Defense
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IANA	Internet Assigned Numbers Authority
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISP	Internet Service Provider
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
MIME	Multipurpose Internet Mail Extensions
MPEG	Moving Picture Experts Group
NCES	Net-Centric Enterprise Services
NPE	Non-Person Entities
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PKI	Public Key Infrastructure
PNG	Portable Network Graphics
RCVS	Reliable Certificate Validation Service
REST	Representative State Transfer
RIA	Rich Internet Application

RPC	Remote Procedure Call
RSS	Real Simple Syndication
SAML	Security Assertions Markup Language
SNMP	Simple Network Management Protocol
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UBL	Universal Business Language
UMA	User Managed Access
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUIDs	Unique User Identifier
WADL	Web Application Description Language
WAV	Windows Wave
WRAP	Web Resource Access Protocol
WSDL	Web Services Definition Language
WS-Management	Web Services Management
XACML	Extensible Access Control Markup Language
XML	Extensible Markup Language

Appendix B

WADL example of the Yahoo Web Search Service

```
<?xml version="1.0"?>
<application xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:srch="urn:yahoo:srch"
  xmlns:ya="urn:yahoo:api"
  xmlns:html="http://www.w3.org/1999/xhtml"
  xmlns="http://research.sun.com/wadl/2006/10">
```

This reuses the same WebSearchResponse.xsd XML representation utilized by the WSDL 2.0 above.

```
  <grammars>
    <include href="WebSearchResponse.xsd"/>
    <include href="Error.xsd"/>
  </grammars>
```

This declares the resource base path which is used in the base URL.

```
  <resources base="WebSearch">
    <resource path="webSearch">
      <doc xml:lang="en" title="Yahoo! Web Search Service">
        The <html:i>Yahoo Web Search</html:i> service allows you to search the
        Internet for web pages.
      </doc
     <method href="#search"/>
    </resource>
  </resources>
```

This declares an HTTP Get operation for the search. The request querystring parameters are explicitly defined under the method.

```
  <method name="GET" id="search">
    <doc xml:lang="en" title="Search Internet for web pages by keyword"/>
```

Notice that the parameters of the querystring are explicitly listed instead of packaging within a XML complex type as done with WSDL 2.0.

```
    <request>
     <param name="query" type="xsd:string" required="true" style="query">
        <doc xml:lang="en" title="Space separated keywords to search for"/>
      </param>
      <param name="results" type="xsd:int" default="10" style="query">
        <doc xml:lang="en" title="Number of results"/>
      </param>
      <param name="start" type="xsd:int" default="1" style="query">
        <doc xml:lang="en" title="Index of first result"/>
      </param>
    </request>
```

Potential responses to the request include an XML result set, a JSON encoded response (note there is no JSON machine description available), and fault code of HTTP 400 which returns an XML error from the error.xsd.

```
    <response>
      <representation mediaType="application/xml" element="srch:ResultSet">
        <doc xml:lang="en" title="A list of search results in XML format"/>
      </representation>
      <representation mediaType="application/json">
        <doc xml:lang="en" title="A list of search results in JSON  format"/>
      </representation>
      <fault id="SearchError" status="400" mediaType="application/xml"
        element="ya:Error"/>
    </response>
  </method>
</application>
```