NATO Communications and Information Agency
Agence OTAN d'information et de communication

AGENCY INSTRUCTION

INSTR TECH 06.02.07

SERVICE INTERFACE PROFILE FOR REST MESSAGING

Effective date:

Revision No: Original

Issued by: Chief, Core Enterprise Services

Approved by: Director Service Strategy

Table of Amendments

Amendment No	Date issued	Remarks

Author Details

Organization	Name	Contact Email/Phone
NCI Agency	R. Fiske	rui.fiske@ncia.nato.int
NCI Agency	A. Ross	
NCI Agency	A. Tucker	

Table of Contents

AGENCY INSTRUCTION 06.02.07

SERVICE INTERFACE PROFILE FOR REST MESSAGING

0 PRELIMINARY INFORMATION

0.1 References

A. NCIA/GM/2012/235; Directive 1 Revision 1; dated 3 May 2013
B. NCIARECCEN-4-22852 DIRECTIVE 01.01, Agency Policy on Management and Control of Directives, Notices, Processes, Procedures and Instructions, dated 20 May 2014
C. NCIARECCEN-4-23297, Directive 06.00.01, Management and Control of Directives, Processes, Procedures and Instructions on Service Management, dated 03 June 2014

0.2 Purpose

This Technical Instruction (TI) provides detailed information, guidance, instructions, standards and criteria to be used when planning, programming, and designing Agency products and services. In this specific case the TI defines a Service Interface Profile (SIP) for one of NATO's Core Enterprise Services.

TIs are living documents and will be periodically reviewed, updated, and made available to Agency staff as part of the Service Strategy responsibility as Design Authority. Technical content of these instructions is the shared responsibility of SStrat/Service Engineering and Architecture Branch and the Service Line of the discipline involved.

TIs are primarily disseminated electronically[1], and will be announced through Agency Routine Orders. Hard copies or local electronic copies should be checked against the current electronic version prior to use to assure that the latest instructions are used.

0.3 Applicability

This TI applies to all elements of the Agency, in particular to all NCI Agency staff involved in development of IT services or software products. It is the responsibility of all NCI Agency Programme, Service, Product and Project Managers to ensure the implementation of this technical instruction and to incorporate its content into relevant contractual documentation for external suppliers.

1 INTRODUCTION

In order to ensure compatibility between services, both within NATO, and between NATO and its partners, there is a need to ensure that a standard (and standards-based) profile can be defined which will be mandatory for all service operations in the NATO Network Enabled Capability (NNEC) messaging environment.

It is recognized that NATO communication and information systems (CIS) operate in a heterogeneous environment, with service providers and consumers operating under multiple different frameworks and application contexts. Therefore, this Service Interface Profile (SIP) has been designed to accommodate these differences, and offer the widest possible support for a messaging infrastructure.

This specification provides the interface control for Representational State Transfer (REST) web services (known as RESTful web services) that are deployed within the NNEC web service

[1] https://servicestrategy.nr.ncia/SitePages/Agency%20Directives%20(Technical).aspx

infrastructure. This covers only the call from a *Web Service Consumer* to a *Web Service Provider* using REST, and the response from the service provider. It includes how the message must be structured and the elements that must be contained within the call.

This profile has evolved in response to the available technologies and mechanisms that can be used to apply messaging within the wider context of the web services environment. Furthermore, it has been tested against the service implementations of NATO and Coalition member nations.

1.1 Scope

REST is an architectural style defined as a set of constraints on a distributed hypermedia system and implemented by a set of standard protocols that adhere to these constraints. We here define the necessary elements of the underlying specifications that REST uses. The document will not present the complete specifications, namely hypertext transport protocol (HTTP), JavaScript object notation (JSON), extensible markup language (XML) or hypertext markup language (HTML). As a result the SIP is not an exhaustive definition of those specifications but rather an extension to further clarify how the REST style can be employed for building web services in an NNEC environment. As a recommendation in case of doubt refer to the correspondent formal specification noted on the references section.

1.2 Audience

The target audience for this specification is the broad community of NNEC stakeholders, who are delivering capability in an NNEC environment, or anticipate that their services may be used in this environment.

These may include (but are not limited to):

- Project Managers procuring NATO communication and information systems.
- The architects and developers of service consumers and providers that interact with the (NATO Unclassified) (NU)) SIP Proposal – REST Messaging v.1.0.
- Coalition partners whose services may need to interact with NNEC services.
- System integrators delivering systems into the NATO environment.

1.3 Notational Conventions

The following notational conventions apply to this document:

- The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [IETF RFC 2119, 1997].
- Words in italics indicate references to terms defined in Section 1.4.
- Courier font indicates syntax derived from the different open standards.

1.4 Taxonomy Allocation

This service falls under the following allocation in the C3 Taxonomy [Tidepedia "C3 Taxonomy", 2012]:

> Technical Services | Core Enterprise Services | SOA Platform Services | Message-oriented Middleware Services.

1.5 Terminology

The following terminology is used in this specification:

Web Service Provider	A service that produces data for other services.
Web Service Consumer	A service or application that calls other services in order to retrieve data.
Message	The structure used for exchanging data between the *Web Service Provider* and *Web Service Consumer*.
Header	The part of the *message* that contains additional information about the message beyond the data that is being exchanged.

1.6 Goal

The goal of this profile is to:

- Support messaging patterns that will allow the broadest possible range of interoperability between systems.

1.7 Non-Goals

The following topics are outside the scope of this profile:

- Define how security may be applied to messages.

1.8 Relationships to Other Profiles and Specifications

1.8.1 Normative References

The following documents have fed into this specification, and are incorporated as normative references. Where more than one version of the same standard or profile is listed, then support for requirements from both versions are covered, although a single service need not implement both versions.

- HTTP 1.1-RFC 2616 (IETF) – http://www.ietf.org/rfc/rfc2616.txt
- Uniform Resource Identifiers (URI) – RFC 3986 (IETF) – http://www.ietf.org/rfc/rfc3986.txt
- Media Types – RFC 2046 (IETF) - http://www.ietf.org/rfc/rfc2046.txt
- XML Media Types – RFC 3023 (IETF) – http://www.ietf.org/rfc/rfc3023.txt
- The application/json Media Type for JSON – RFC 4627 (IETF) – http://www.ietf.org/rfc/rfc4627.txt
- WSDL 2.0 (W3C Recommendation) – http://www.w3.org/TR/wsdl20.

2 BACKGROUND

2.1 REST Definition

REST is an architectural style defined by Roy Fielding in his doctoral thesis [Fielding, 2000]. Fielding derived a number of architectural styles by successively introducing constraints on how a distributed system can be characterized. The motivation for the REST style is to characterize the World Wide Web distributed hypertext system in order to highlight the architectural features which lead to such a highly successful distributed computing platform. The architectural constraints defining REST which concern this SIP are briefly as follows:

- Client-server: separation of concerns allows components to evolve independently.
- Stateless: communication from client to server must ensure that all information necessary to understand a request is included in the request; it cannot rely on the context of the communication or state of the server.

- Cache: server responses can be annotated as cacheable and clients can take advantage of previous responses to avoid repeated equivalent requests.
- Uniform interface: the principle of generality ensures that all components use the same generic interface, optimized for the common use cases. This constraint is a combination of four interface constraints:

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

Since REST is derived by essentially reverse engineering the World Wide Web standards, it is natural that the standard implementation of the REST architecture is based on precisely the World Wide Web standards of HTTP, HTML and URI along with their respective dependent standards. As such, we can consider the REST specification as the combined open standard specifications of the underlying protocols and data formats.

2.1.1 RESTful Web Services

The REST architectural style can be employed for implementing web services which are known as "RESTful web services". RESTful web services rely on HTTP as the transport protocol between *Web Service Providers* and *Web Service Consumers* using the HTTP verbs GET, PUT, POST, DELETE etc. in their specified manner. Resources that are exposed to RESTful web services are identified by URI and are primarily represented to *Web Service Consumers* in XML or JSON.

Note that this definition of RESTful web services also encompasses the existing World Wide Web where the end-user is the *web service consumer* using a web browser. It is the interaction between the user and the web browser which RESTful web services aim to make amenable to automatic processing, and as such the focus of RESTful web services is on making the web service response machine interpretable.

The World Wide Web is an existence proof of a highly successful, scalable networked architecture which can accommodate all manner of applications and use cases. There are readily available programmatic tools that use the same protocols and interfaces as the user-driven web browsers and so can be used to programmatically interact with these existing applications.

2.2 Considerations for the Use of REST

Distributed computing is hard. The history of computing highlights repeated attempts to address the issues of latency, distributed state, global address spaces, concurrency and partial failure with various schemes to make it easier for the software engineer to build robust and scalable distributed systems. One commonly repeated approach is to abstract away the differences between local and remote computing such that the software engineer need not be concerned with the location of the objects or services being invoked, only on their interfaces. This approach, typically using remote procedure calls (RPC) with a standardized interface definition language and object/service discovery and management, has been proposed a number of times: Sun's RPC, the Open Software Foundation's DCE (distributed computing environment), Microsoft's DCOM (distributed component object model), OMG's CORBA (common object request broker architecture), Java's RMI (remote method invocation), XML-RPC and most latterly with industry consortia proposals for simple object access protocol (SOAP), WSDL and the WS-* stack. However, it can been argued ([Sun Microsystems Laboratories, Inc., 1994]) that remote computing is inherently different to local computing: it is much more difficult to manage resources globally; latency and caching are far more important for remote objects; partial failure and concurrency are much harder to deal with and less deterministic remotely

than locally and so new techniques must be brought in to deal with transactions, reliable messaging etc.

The REST approach, in contrast, distinguishes between local and remote computing and focuses on ways to handle the specific issues inherent in a distributed computing environment. The issues of latency and caching are addressed by explicitly ensuring that responses can be cached wherever possible. The issues of distributed state, concurrency and partial failure are managed by ensuring that methods which do not change the remote state are explicitly marked as idempotent – i.e. the method can be repeated with the same result – and that the state of a resource is bounded in any communications between client and server in such a way, that state is not part of the context of the communication. In addition, a server can explicitly flag various different types of failure as well as enumerate the allowed transitions of a resource in a response. Finally, a global addressing scheme is defined allowing resources to be identified generically on the network.

Since REST is an architectural style, it is directly comparable with service-oriented architecture (SOA). While both styles of architecture are characterized by having loosely coupled, modular components, the SOA approach in contrast to the REST approach stresses the location transparency of these components, and so also requires that components be discoverable at run time and that their interfaces be described in such a way that they are also usable at run time (referred to as dynamic binding). These differences in the characteristics of the REST and SOA architectural styles are highlighted by differences in the way they are most normally implemented.

For SOA, SOAP-over-HTTP and WSDL are the current standards of choice for protocol and service description respectively. WSDL documents describe, in machine-readable form, the syntax of the operations and the data types made available by a given service. As such, services described using WSDL are predominantly RPC-style services having a service-specific interface and an endpoint URI at which that interface is exposed. Services are discovered at runtime by way of a service registry that can be programmatically queried for a specific interface, or other service characteristic.

For REST, HTTP and URI are the standards for protocol and resource identification respectively. Since HTTP defines a uniform interface using the GET, POST, PUT, DELETE etc. "verbs", there is no additional requirement for interface definition of any specific service – all services use the same interface methods. In contrast, each piece of data to be operated on in a RESTful web service has its own URI. A RESTful web service is therefore described by declaring a set of URI "nouns", one for each piece of data, and by specifying what the HTTP "verbs" mean when applied to those nouns. Services are discovered in much the same way that web pages are discovered: by following URI links in the responses to previous HTTP requests.

In practice, the differences between the REST and SOA styles are often ignored and a large number of currently deployed web services sit somewhere between being purely RESTful and purely RPC. These types of web service have been described as REST-RPC hybrid (see [Leonard and Ruby, 2007]), and tend to use ad-hoc methods outside of the HTTP protocol, for example by adding bespoke method names to URIs and simply using HTTP's GET or POST methods arbitrarily. REST-RPC hybrid web services tend to be described by declaring the set of URIs to be used as a combination of "noun" and "verb" such that one URI will be used to create a resource while another will be used to update it and yet another to delete it etc. In as far as these types of web service are quite common, a *Web Service Consumer* should be prepared to handle REST-RPC hybrid web services, but this SIP does not make recommendations about their use as a *Web Service Producer*.

Ultimately, the choice of which architectural style to follow, and therefore which technologies to use, should be driven more by the type of service interaction which is envisaged in a given system rather than the particular implementation of a service. In addition, the choice of architectural style need not be restricted to exclusively REST or SOA, but could equally include both styles in a larger system, or

include other styles such as asynchronous messaging. Some high-level guidance for the choice of implementation is:

- Use SOAP/WS-* where the service interaction is fundamentally RPC, for example when codifying some business process.
- Use message queuing implementations where the service interaction is fundamentally around asynchronous messaging.
- Use REST/HTTP where the service interaction can be modeled naturally using named resources (nouns) along with the generic uniform HTTP interface methods to cover the majority of the use cases.

2.3 XML and JSON

RESTful web services require that an HTTP response is machine-readable but are also much easier to work with if the response is also human-readable. Both XML and JSON text formats meet these requirements, with JSON being preferred for the more lightweight *Web Service Consumers*. XML tends to be preferred where messages need to be validated, filtered, signed, transformed etc. by intermediaries, since there is a well-established family of specifications dealing with XML.

An important characteristic of REST is that a service response should be able to describe the operations which can be performed on the service's resources – this is the notion of "hypermedia as the engine of application state" in the definition of REST. Both XML and JSON formats are quite capable of representing URI strings, with XML Schema offering this as a fundamental data type.

2.4 Best Practices

2.4.1 PUT and POST

2.4.1.1 CRUD

Create, read, update and delete (CRUD) are the main operations used when dealing with information in persistent storage, most usually seen in relational database systems where they correspond directly to the structured query language (SQL) operations INSERT, SELECT, UPDATE and DELETE respectively.

While REST/HTTP has similar operations, the correspondence with CRUD is not a direct one-to-one match, specifically for create and update methods, but also due to the granularity of HTTP resources compared to typical entities in a database system and how transactions are processed.

In HTTP, the creation and updating of resources can be done using either the PUT or POST methods, where the choice of which method to use depends on how the resource is referred to in the HTTP request, whether the new or updated resource URI is determined by the server or client and whether multiple invocations of the same method will result in the same end-state of the resource, i.e. whether the method is idempotent or not.

2.4.1.2 The HTTP PUT Method

The PUT method can be used by a client to create or update a resource where:

- The request URI refers to the resource to be created or updated, e.g. PUT /a/1 refers to the /a/1 resource directly.
- The client can determine the URI of the resource by itself in advance, for instance by being given the URI by the server in a previous interaction.
- The body of the HTTP request contains the full representation of the state of that resource.
- Repeating exactly the same request more than once has exactly the same effect as making the request only once.

Note that the server will distinguish between creation and update of the resource by the response code returned. The server must return HTTP response code 201 (Created) if a new resource is created. Otherwise, the server should return code 200 (OK), 204 (No content) or some other response code as necessary, but not 201 (Created).

Since the PUT method is idempotent, an intermediary is allowed to cache the response to a given request in order to reduce network latency and traffic. In addition, a client can safely repeat a PUT request where it is unknown whether the request made it to the server, for instance if a network timeout occurred.

2.4.1.3 The HTTP POST Method

In contrast, the POST method is used by a client to create or update a resource where:

- For creation, the request URI is not the URI of the resource to be created, but instead a "container" or "factory" resource, e.g. POST /customers might be used to create a new customer resource contained by the /customers resource. Note that the request URI must be an existing resource on the server, otherwise an error is returned.
- The server is responsible for determining the URIs of the resource to be created and as such these URIs are managed directly by the server.
- For update, a POST request to an existing resource is expected to contain information in the body of the request as to what updates should be made to the given resource or subordinate resources. For example, sending POST /log might be used to append an entry to a log resource without creating a new subordinate resource.
- Repeating the same request more than once can result in more than one resource being created or resources being updated twice, for instance adding the same log entry twice.

In response to a POST, if a new resource has been created then the server should respond with code 201 (Created) and a Location header giving the new resource URI. In other cases, the server can respond with code 200 (OK) or 204 (No content) or some other code, depending on what needs to be conveyed back to the client.

The POST method is not idempotent and the server response to a given client request should not be cached, unless the response includes specific Expires or Cache-control headers.

From [IETF RFC 2616, 1999], the POST method is intended to cover the following use cases:

- Annotation of existing resources
- Posting a message to a bulletin board, news group, mailing list, or similar group of articles
- Providing a block of data, such as the result of submitting a form, to a data-handling process
- Extending a database through an append operation.

In general, the request URI used in a POST request is a resource which is considered to be a collection or factory resource, and the body of the POST request is considered to be a subordinate entity to that collection or factory resource. Again, from [IETF RFC 2616, 1999], a resource is subordinate to another resource "in the same way that a file is subordinate to a directory containing it, a news article is subordinate to a newsgroup to which it is posted, or a record is subordinate to a database."

2.4.1.4 Granularity of a Resource

The resources exposed by a RESTful application tend to be of a larger granularity than data items in a traditional database, driven by the way resources tend to be used by clients and performance requirements when transferring resource state over a network. Since the network latency of a set of HTTP interactions is generally more of a concern than when using a local database, those resources

whose state is persisted in a relational database will tend to consist of a number of related data items from the database combined in to a single logical and self-contained representation.

This is another reason that the CRUD operations on a database do not necessarily map directly onto the REST/HTTP architecture – in general there is not a one-to-one map between a row in a database table and a resource at a given URI.

2.4.1.5 Transactions

Transactions can play an important role in database-driven applications, ensuring that the overall application state remains consistent over a series of interactions. However, for RESTful applications, none of the standard HTTP methods can be used to lock resources, begin, commit or roll back transactions. The general technique in a RESTful architecture for accommodating methods which do not map directly to the standard ones is to instead come up with a new class of resources which encapsulate the concept in such a way that the standard HTTP methods can apply. In much the same way that the adage, "there ain't a noun which can't be verbed", applies in English, in REST, there ain't a non-REST verb which can't be turned into a REST resource noun.

In the case of transactions, a RESTful architecture would coin a set of transaction resources either as subordinates to a resource they refer to, or as subordinates to some transaction manager resource, and then ensure that the state of the transaction resource is consistently linked to the resources it manages. See [Leonard et al., 2007] for examples of how transactions can be modeled as resources in their own right.

2.4.1.6 POST once exactly

One way to cope with the issue of POST potentially altering a resource in a non-deterministic way has been proposed in [Nottingham, 2005], and while not having been accepted as an RFC, has been successfully used (e.g. IBM's System Director REST application programming interfaces (API) to avoid repeated POST requests having undesired side-effects.

The general approach is that a server informs a client of special "POE" (post once exactly) resources which the client can then perform a POST to. The contract of a POE resource is that the first time a client performs a POST, the server will respond with an affirmative 2xx status code, but then if a client performs another POST to the same resource, the server should respond with a 4xx code, e.g. 405 (Method not allowed).

2.4.1.7 HATEOAS

The phrase "Hypertext as the engine of application state", abbreviated as HATEOAS, refers to a core property of RESTful architectures, namely that a server should be able to tell its clients which transitions in application state can be taken at any point. By responding with resource representations which contain references to other resource URIs, a service can encapsulate its domain knowledge and contractual understanding of what can be done to the resources it manages. In this way, the coupling between client and server can be reduced, such that a client need not be hard-coded with a pre-existing understanding of which resources to manipulate and what methods to apply to them, but can instead "follow its nose" to the URIs it receives from the server.

Responding with URIs as part of the representation of resources is important when using the PUT method to create resources, as the client must use the URI of the resource to be created as the context of the PUT method.

2.4.1.8 PUT and POST Summary

The main differentiation between the HTTP methods PUT and POST is that PUT is idempotent while POST is not; so it is good practice to use PUT for the creation and update of resources where possible, leaving POST for those cases which do not fit quite as neatly into the semantics of PUT.

Using PUT for create and update does however require that the client knows in advance the URI of the resource it will be manipulating, which either results in a more tightly coupled client, or additional information from the server from previous responses. The client must also PUT a complete representation of a resource, even when it is updating only a small part of that resource. In some cases it is, therefore, easier and may take fewer network interactions for the client to POST to a container or factory resource and if necessary retrieve a new URI in one go.

2.4.2 Uniquely Identifiable and Network-addressable Resources

In the context of REST addressability means ensuring that applications expose their useful data as resources and give them URIs. Since the URI encodes the protocol (scheme), host and port (authority) and path to a resource, a machine can automatically make use of the resource without having to be given separate instructions on how to make use of the address.

A URI identifies a resource rather than a representation. When a client asks a server for a resource, the server will respond with the best possible representation for that resource, given the client's preferences. A client can express its preferences by using the HTTP Accept header.

Every URI should designate exactly one resource. Note that this does not mean the converse is true – that every resource has exactly one URI – as in general there is one URI per resource and potentially many representations per resource. In some cases, however, the same resource can have more than one URI; this is termed URI aliasing. For instance, the most current version of a document might have two URIs: one with the date of publication and another with the string "latest" as part of its resource URI. However, if a resource can have multiple addresses, then any other resources which relate to it will also need to either relate to all the possible addresses of the resource, or arbitrarily pick one URI, or use one canonical URI of the resource. Giving multiple addresses to a resource dilutes the value of a resource – the network effect suggests that the "value of a resource can be measured by the number and value of other resources in its network neighbourhood" (Metcalfe's law), so by having multiple addresses this potentially splits the network neighbourhood into isolated islands of connected resources.

Specific representations or a resource can have their own URI, for instance an image resource might have a canonical URI of http://example.com/images/tree along with alternative URIs http://example.com/images/tree.svg and http://examples.com/images/tree.jpeg etc. In this way, a client requesting the canonical URI with a preference for JPEG (Joint Photographic Experts Group) imagery might result in the server responding with a 303 (See other) pointing to the URI of the JPEG representation.

Making use of the fact that the fully qualified domain name system (DNS) name for a server is a globally recognized address for the server, a URI can itself be a globally addressable resource. DNS also allows for global delegation of domains from top-level domains down through counties, organizations and organizational units, where management, ownership and ultimately trust can be delegated in a natural way through the enterprise. This natural delegation should be used wherever possible by services in order to publish and manage their own resource under their own delegated domain name, i.e. services should publish their resources using fully qualified DNS names in the authority section of the URI rather than local names or numeric Internet protocol (IP) addresses.

Often, a number of RESTful services may run on a server in separate processes on separate ports and under different paths, or for the purposes of load-balancing and redundancy, on completely separate servers. This can lead to a fragmented address space for resources which would conceptually and logically be considered to be at the same server address. For a standard user facing web sites, this is often overcome by the use of a reverse proxy, acting as a front-end to the various services, rewriting incoming URIs, passing on requests for those URIs, and in some cases re-writing URIs in the

responses if the services cannot do so themselves. The proxy essentially maps internal URIs from different servers, ports and paths into a single, consistent external URI space rooted at an externally addressable URI. For RESTful web services, a similar approach can be useful: a reverse proxy can be used as a lightweight method to de-couple the external facing URI address space from any internal changes which might be necessary as the services evolve or scale. A reverse proxy can also help when managing firewalls and security mechanisms, since the proxy becomes the single entry point to the internal network of servers.

2.4.3 Structuring Resources and Relationships between them

In its most general form, a URI is a string of characters comprising a scheme, authority, path and optional query and fragment. For HTTP URIs:

- The scheme is the string "http".
- The authority is the domain name and port number where the HTTP service can be found.
- The path is a hierarchical space of strings separated using the slash (/) character.
- The query is a set of keyword value pairs to be used to parameterize a resource.
- The fragment is used only by the client to refer to a specific part of a resource; this part of a URI is not sent to the server in a request.

One method for expressing the relationships between resources is to make use of the hierarchical nature of the generic URI syntax – that is, the path, query and fragment parts. This hierarchy is meant to express the "order of decreasing significance, from left to right" ([IETF RFC 3986, 2005]). In practice, this hierarchy is most often used to express the relationship of containment, and goes hand in hand with the use of relative URIs which allows resources to link to each other in a local tree representation such that the tree of resources can easily be moved without breaking all the relationship links between local resources. The hierarchy notion is also used to express subordinate resources: one resource which manages other resources "underneath" it in the hierarchy. Subordinate resources can be used as described earlier with the POST method.

When information resources are not related to each other hierarchically, separator characters other than the slash (/) can be used to delimit parts of a URI. Common practice is to use commas (,) when the ordering of the parts matters, and semicolons (;) when the ordering does not matter.

Making use of hypertext is one of the key characteristics of REST, and in this context means that a resource's representation must be able to express relationships to other resources such that a client can follow these relationships and operate on other resources as directed by the server. A relationship between resources can be described using a pair of URIs, denoting the subject resource and the object resource respectively. Commonly, neither the subject resource nor the type of the relationship are made explicit in the representation, although when they are the standard representation is to use a URI to represent the type of relationship and so form a triple of subject, relationship-type and object URIs as described in [W3C RDF-Primer, 2004].

The subject resource is essentially the resource as requested by a client, modulo any redirections the server suggests, and is more commonly referred to as the base URI of a document. However, as soon as the representation of a resource is used outside of the context of a client's request and the server's response, this implicit base URI is lost – for instance if the resource's representation is saved to disk, or sent by a different messaging protocol to another service. The base URI of a resource is also important when using relative URIs, as above, as the algorithm for resolving relative URIs depends on the base URI. For these reasons, common practice suggests that a resource's representation should contain its own URI where possible: for XML, this can be achieved using the xml:base attribute on the document element; for JSON there is no standard, but commonly a JSON object is returned with a special field, "href", used to represent the base URI.

The representation of a resource should also describe that resource's relationships to other resources by using URIs. For XML based representations, the XLink specification [W3C XLink 1.1, 2010] can be used to express these relationships in a natural way. For JSON, current practice is to express the relationship as a JSON object of the form:

```
{"relationshipName": {"href": "http://example.com/a/b"} }
```

where "relationshipName" is a chosen name for the relationship.

2.4.4 Opaque URIs

A fundamental axiom of the architecture of the World Wide Web is that URIs should be opaque to clients. That is, a client should not need to pick a URI apart to determine what it means or what to do with it; the fact that a URI ends with a particular string, for instance ".html" should not be used by a client to deduce that the resource is necessarily an HTML document. Opaque URIs are used to ensure that a server can direct a client to which resources to use and how they should be used without the client having to know in advance the particular URI structure that a server uses; the opacity axiom goes hand in hand with the use of hypertext as the engine of application state and ensures that clients can be generic and loosely coupled from the specific implementation of a server. Where metadata about the resource needs to be conveyed, it is done using the standard HTTP headers and the rest of the information a resource conveys is carried in the representation of the resource itself.

Ensuring resource URIs are opaque to a client does not mean that URIs should be mangled or obfuscated such that they cannot be used to infer any meaning. It only means that a client should not rely on the structure of a URI. Indeed, having some meaning that a human can derive from a URI helps in debugging the intention of the URI in the same way that the HTTP protocol, XML and JSON are all human-readable, and using human-readable identifiers helps convey the intended meaning of the protocols and data to the people developing the services.

If a client is not meant to infer structure from a URI, then it is the responsibility of the server to give the client all the information it needs to operate on the resources it manages, which includes the responsibility of "minting" URIs for the creation of new resources and passing those URIs to the client for use in any PUT or POST methods.

2.4.5 Caching

[IETF RFC2616] describes the basic design behind HTTP caching as well as providing detailed specifications of the HTTP methods, HTTP headers and HTTP response codes required for HTTP caching. However, there is no doubt that the concept of HTTP caching is complex and as such is misunderstood by implementers resulting in potentially harmful results. So why implement HTTP caching? There are two main benefits for providing HTTP caching:

5. Latency – HTTP request/response latency can be reduced by having a representation of the resource closer to the *Web Service Consumer*. This improves the performance as the time to satisfy the HTTP request is reduced.
6. Bandwidth – Network bandwidth can be limited by reducing the number of network hops required in the act of returning a representation in an HTTP response.

To support these two HTTP caching benefits the following HTTP caching mechanisms should be deployed:

1. Expiration caching
2. Validation caching.

Expiration caching supports improved network latency by determining if a cached resource representation is fresh (not out of date). This type of caching can be controlled by two methods:

1. `Expires` header and Date header, or
2. `Cache-Control` header with the following directives: `max-age`; `max-fresh`; and `max-stale`.

In the case where both methods are applied, the `cache-control` directive shall be used.

Not all HTTP verbs can benefit from caching. The HTTP PUT, DELETE AND POST verbs, in their nature, invalidate the resource representation identified by the Request-URI or the Location header. Caching shall primarily be supported by the HTTP GET and HEAD verbs. In the case where best practices are followed, RESTful *Web Service Consumers* that perform an idempotent HTTP PUT request may indicate that the response is cached by including a `cache-control` header with the required directive.

HTTP requests that contain query strings within the Request-URI path shall not be cached. Such requests may trigger side-effects to the resource potentially resulting in a non-consistent cache.

Validation caching supports improved network bandwidth by conditional requests between entities in the HTTP request/response chain to determine if the representation of the resource has changed. This type of caching can be controlled with by two methods:

1. `Last-Modified` header and `If-Modified-Since` header or `If-Not-Modified-Since`, or
2. `ETag` header and `If-None-Match` or `If-Match`.

In both cases if the resource representation has not changed, the *Web Service Provider* shall return a 304 HTTP status code. This response may contain additional `cache-control` header and directives. In the case where the resource representation has been changed, the *Web Service Provider* shall return a representation of the resource that has been requested with a 200 HTTP status code.

With regards to caching it is recommended to use the Last-Modified header as computationally the ETag header may offer more computational work on the server side. However, it must be noted that the ETag header can be used to benefit RESTful web services where multiple *Web Service Consumers* are interacting with a single resource. A good design for ETag header field values can assist with maintaining and achieving consistent resource state.

In order to realize the benefits of HTTP caching we must first understand the potential threats posed by implementing HTTP caching. Firstly, RESTful web service implementations can have a number of intermediaries or HTTP caches between the *Web Service Consumer* and *Web Service Provider*. A HTTP cache is the temporary storage for representations of resources. As a request for a resource representation is made, that request may transit through a number of different types of HTTP caches until it reaches the *Web Service Provider* (in the case where cache conditions have not been met) that is serving the request. There are three main types of HTTP caches:

1. Local HTTP cache – A cache serving a single Web Service Consumer. This type of cache is typically stored on the same computer as the Web Service Consumer and can store the cache in memory hence being capable of serving frequently accessed resource representations immediately.
2. Proxy HTTP cache – A cache serving multiple Web Service Consumers and Web Service Providers. This type of cache is typically located at the organization boundary.
3. Reverse HTTP cache – A cache serving a single *Web Service Provider* for multiple *Web Service Consumers*.

With the potential for there being many types of HTTP caches between a *Web Service Consumer* and a *Web Service Provider*, there is the high possibility of inconsistency between the representations of a resource along the HTTP request/response chain. This is harmful as a *Web Service Consumer* may be acting on stale representations. *Web Service Providers* may be able to notify HTTP caches that a resource representation has been modified or deleted; however, it must be noted that this goes against one of the main REST principles for stateless communications between *Web Service Consumers* and *Web Service Providers*.

Secondly, NATO must provide effective and efficient conduct of modern joint military operations where cross-domain information exchange is required between different classification security domains within NATO; NATO and NATO nations; NATO and NATO-led missions; NATO and non-NATO nations; and, NATO and other government departments or non-government organizations. As such, access requests for resource representations must be robustly controlled in order to preserve the confidentiality and integrity of those resources. Resources that are deemed to be sensitive shall not be cached.

Thirdly, HTTP caching is only beneficial if it proves to save network latency and network bandwidth. If resources change frequently, then HTTP caching provides no benefits for RESTful web services.

To support the second and third threats to the benefits of HTTP caching *Web Service Consumers* and *Web Service Providers* can indicate that caching is not required/permitted for that resource representation. A *Web Service Consumer* shall indicate to all entities in the HTTP request/response chain that information shall not be cached by inserting the HTTP header `cache-control` with the additional directive of `no-store`. The `no-cache` directive may optionally be used; however, this shall not be recognized by HTTP 1.0 implementation. In the case where the *Web Service Provider* shall respond with a request for sensitive information, the HTTP response shall contain the HTTP header `cache-control` with the additional directive of `no-store`. The `private` directive may be used; however, a private cache may cache the response. Additionally, a *Web Service Provider* can include an `Expires` header with either an invalid date value or a date value that is the same as the `Date` header field value.

3 SERVICE INTERFACE SPECIFICATION

3.1 Service Interface

As stated earlier, REST is an architectural style defined by the constrained and consistent use of a number of protocols. For that reason there is no single defined service interface other than the use of HTTP itself. However, there are machine-processable description languages for HTTP-based web applications using REST. These descriptions provide information on how a RESTful web service can be invoked, which parameters are expected, and what data structures are returned by the service.

3.1.1 Definition

In order to increase interoperability, REST services SHALL have a clearly defined interface.

The interface SHALL be clearly documented in human-readable format describing:

- All resources
- Methods supported for each resource
- Internet media types and representation formats supported in HTTP requests and responses
- All fixed URIs
- Query parameters used for URIs
- URI templates and rules for token substitution

- Supported security mechanisms for accessing resources
- Relationships between resources and type of relationship.

The interface SHOULD be defined in machine-readable format, using Version 2.0 of the WSDL.

WSDL 2.0 [W3C WSDL 2.0: Primer, 2007] is the most recent update to an earlier specification that now accepts bindings to all HTTP request methods and is a W3C (World Wide Web Consortium) recommendation. While this results in better support for RESTful web services, it is recognized that the support for WSDL 2.0 in terms of tools is weak compared to the older Version 1.1.

Note that by following the best practices for REST, a service already has a well-defined interface specification in HTTP along with the ability to specify, in machine-readable format, how a client can use its resources by following the HATEOAS pattern. In addition to this, typical interface definition languages such as WSDL are intended to cover the RPC style of service interaction rather than the resource-oriented REST style. As such, the emphasis for service interface definition is placed first on using the REST best practices, then on describing the interface in a human-readable format as above, followed by the use of WSDL 2.0 where applicable.

3.2 Transport

Although the REST architectural style could conceivably be used over protocols other than HTTP, these are not covered by this SIP. Therefore, all representations MUST be transferred using HTTP 1.1 ([IETF RFC 2616, 1999]).

3.3 Message Structure

3.3.1 Input

This SIP places no constraints on the type of data that can be sent to the *Web Service Provider* in the body of an HTTP request. However, it is RECOMMENDED that only XML or JSON be used. The requested data object representation SHALL be negotiated as described in Section 3.3.1.4.

3.3.1.1 HTTP methods

REST is heavily reliant on the uniform use of HTTP verbs. All HTTP verbs apply to the request URI entity which is the URI specified on the HTTP request.

RESTful web services SHALL use the HTTP verbs as specified in HTTP 1.1 [IETF RFC 2616, 1999].

It is further RECOMMENDED that RESTful web services use the prescribed HTTP verbs for CRUD operations as specified in Table 1.

Table 1
RECOMMENDED HTTP verbs supporting CRUD operations in REST

Verb	Description
GET	Retrieves an information object identified by the request URI.
POST	Updates an information object identified by the request URI. The request URI MAY: create new additional information objects; update additional information objects; or perform a variety of create or updates of information objects.
PUT	Creates a new information object identified by the request URI. Updates an information object identified by the request URI. It is RECOMMENDED that the update operation is a complete update of the information object identified by the request URI.
DELETE	Deletes an information object identified by the request URI.
HEAD	Retrieves the same HTTP header fields and HTTP status code as the GET HTTP verb without the representation of the information object identified by the request URI.
OPTIONS	RESTful web services MAY use this HTTP verb to determine the list of HTTP verbs supported by the information object identified by the request URI.

3.3.1.2 HTTP headers

HTTP headers MUST only be used to transmit representation metadata and message control information.

Parameters for the operation to be performed on the resource MUST NOT be sent as HTTP headers.

It is RECOMMENDED that only standard headers are used. An informational list of these HTTP headers for IANA registry header fields can be found in [IETF RFC 4229, 2005].

However, well-documented custom headers MAY be used, for example in order to prevent cross-site scripting attacks.

The set of RECOMMENDED and REQUIRED request headers are included in Table 2.

Table 2
Request HTTP headers

HTTP header field name	Required	Notes
`Accept`	RECOMMENDED	This specifies which media types are acceptable in the response (see Section 3.3.1.4).
`Cache-Control`	OPTIONAL	This is used to specify how all entities will cache the information. For further details on this and other caching headers, see Section 3.3.1.6.
`Content-Length`	RECOMMENDED	This specifies the length of the body of the message (if known).
`Content-Type`	REQUIRED	This specifies the format of the body sent in the request.
`Cookie`	OPTIONAL	This contains information that can be used for preserving state across calls (see Section 3.3.1.5).
`Date`	OPTIONAL	This specifies the date and time of the submitted HTTP request.
`Host`	REQUIRED	This identifies the host and port to which the request is being sent.

3.3.1.3 URIs

A URI provides a simple and extensible means for identifying an information object on the network. The syntax and semantics of a URI SHALL be represented as specified in [IETF RFC 3986, 2005].

Each information object, on the network, SHALL be globally addressable.

3.3.1.3.1 URI Scheme Name

The URI `scheme` name SHALL be `http` as defined by [IETF RFC 2616, 1999] or `https` as defined by [IETF RFC 2817, 2000].

3.3.1.3.2 URI hierarchical component

The URI `hierarchical` part, representing the naming authority, SHALL contain a `host` subcomponent.

The `host` SHALL be globally addressable on the network.

It is RECOMMENDED that the `host` is identified by a fully qualified domain name (FQDN; refer to Domain Name System [IETF RFC 1035, 1987]).

The `host` MAY be identified by an IPv4 address, however, URIs containing such a `host` representation SHALL not be permitted in cross-domain information exchange scenarios.

3.3.1.3.3 URI Path

The `path` component of the URI contains the scoping information and SHALL be persistent.

It is RECOMMENDED that the URI `path` contains versioning information of the information object to support persistent URIs on the network.

In the case where persistence cannot be supported, the *Web Service Provider* SHALL support HTTP redirects with the HTTP `Location` header field.

To indicate hierarchical relationships between information objects it is RECOMMENDED that the URI `path` is constructed using forward slash separators ('/').

To indicate non-hierarchical components within the URI `path` it is RECOMMENDED to use a comma (',') or semi-colon (';').

A URI `path` MAY contain a query part.

A URI `path` containing a query part SHOULD use an ampersand ('&') to separate parameters.

3.3.1.3.4 Security

A URI SHALL be opaque to all consumers.

Web Service Consumers SHALL NOT be capable of gathering sensitive information about the information object or the CIS containing the information object through aggregation techniques carried out on the URI.

3.3.1.3.5 Further Constraints

Web Service Providers SHOULD NOT use file type extensions within the URI to provide the representation of an information object.

Web Service Consumers and *Web Service Providers* SHOULD support content negotiation as specified in Section 3.3.1.4.

The timestamps in used headers or queries SHOULD be formatted according to the [ISO 8601:2004, 2004] standard.

3.3.1.4 Information Object Representation

The *Web Service Consumer* SHALL encode the HTTP `Accept` header field value with one or more of the supported Internet Media Types (as specified in [IETF RFC 3023, 2001]).

The *Web Service Provider* SHALL honour the preferred Internet Media Type requested by the *Web Service Consumer* or return a `406` (not acceptable) HTTP status code.

Figure 1 illustrates supporting representations of information objects using the HTTP `Accept` header field (specified in Table 2) as the RECOMMENDED content negotiation mechanism.

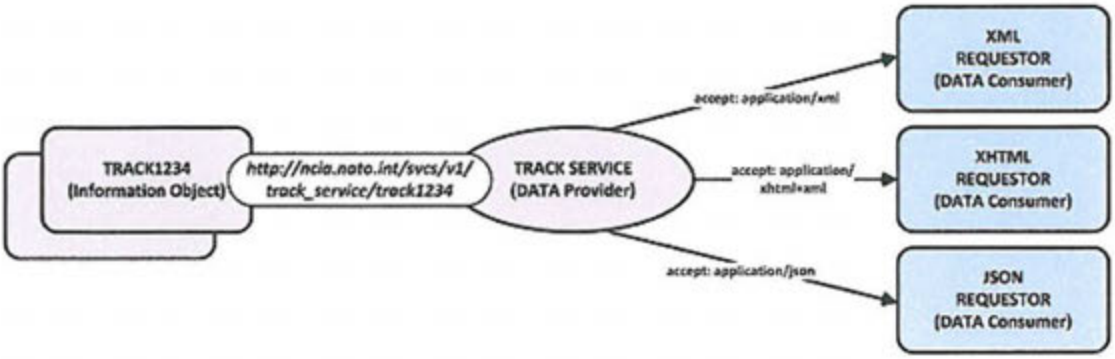

Figure 1 RECOMMENDED mechanism for supporting information object representation

3.3.1.5 Cookies

According to [Fielding, 2000] "Cookies ... violate REST because they allow data to be passed without sufficiently identifying its semantics". Essentially, cookies are used to maintain state, which is in violation of the principles of REST.

However, this SIP recognizes that cookies are widely used to contain session and authentication data. Therefore:

- Cookies SHALL be used in accordance with [IETF RFC 2965, 2000].
- Cookies MAY be used to contain session data.
- Cookies MAY be used to contain authentication data.
- Cookies MAY NOT be used to contain other data.

3.3.1.6 Caching

It is RECOMMENDED that *Web Service Consumers* support caching.

Web Service Consumers that claim support for caching SHALL support caching for the HTTP GET and HEAD verbs only.

A URI path that contains a query element SHOULD be treated as un-cacheable.

Web Service Consumers and *Web Service Providers* on the network SHOULD have synchronized time sources.

3.3.1.6.1 Expiration Caching

Web Service Consumers that claim support for caching can indicate to the *Web Service Providers* whether to return a cached response and the threshold for the age of the response the *Web Service Consumer* is willing to accept.

Table 3 specifies the HTTP header fields and, where appropriate, header field values that are REQUIRED and RECOMMENDED to support expiration caching.

Table 3
Specifications for HTTP Request header fields required for expiration caching

HTTP header field name/ (header field value)	Description
`Cache-Control/ (max-age)`	RECOMMENDED header field and header field value indicating that the client is willing to accept a response whose age is no greater than the specified time in seconds.
`Cache-Control/ (min-fresh)`	RECOMMENDED header field and header field value indicating that the client is willing to accept a response whose freshness lifetime is no less than its current age plus the specified time in seconds.
`Cache-Control/ (max-stale)`	OPTIONAL header field and header field value indicating that the client is willing to accept a response that has exceeded its expiration time.

3.3.1.6.2 Validation Caching

Validation caching allows the *Web Service Consumer* to interrogate the *Web Service Provider* for the purposes of determining if the cached HTTP response is still valid.

Table 4 specifies the HTTP header fields and, where appropriate, header field values that are REQUIRED and RECOMMENDED to support validation caching.

Table 4
Specifications for HTTP Request header fields required for validation caching

HTTP header field name/ (header field value)	Description	Conditions
`Cache-Control/ (only-if-cached)`	OPTIONAL header field and header field value	This header field and header field value are RECOMMENDED to be set in tactical networks where links are impoverished.
`If-Modified-Since`	OPTIONAL header field	RECOMMENDED header field when the cached response contains the `Expires` and `Last-Modified` header fields. This header field MUST NOT be set if the HTTP request contains the header field `If-Unmodified-Since`.
`If-Unmodified-Since`	OPTIONAL header field	RECOMMENDED header field when the cached response contains the `Expires` and `Last-Modified` header fields. This header field MUST NOT be set if the HTTP request contains the header field `If-Modified-Since`.
`If-Match`	OPTIONAL header field	RECOMMENDED header field when the cached response contains the `ETag` header field. This header field MUST NOT be set if the HTTP request contains the `If-None-Match` header field.
`If-None-Match`	OPTIONAL header field	RECOMMENDED header field when the cached response contains the `ETag` header field. This header field MUST NOT be set if the HTTP request contains the `If-Match` header field.

3.3.1.6.3 Security

Caching of sensitive information by *Web Service Consumers* SHALL be prohibited.

Information SHALL NOT be cached when an HTTP request contains a HTTP `Cache-Control` header field with the values: `no-store` and `no-cache`.

Additional constraints, such as digital encryption and digital signatures, SHOULD be applied in order to provide additional protection for confidentiality, availability and integrity of sensitive information objects.

3.3.1.7 Reserved Parameters

Currently this SIP does not reserve any parameters for exclusive use.

3.3.2 Output

This SIP places no constraints on the type of data that can be sent by the *Web Service Provider* in an HTTP response. However, it is RECOMMENDED that only XML or JSON be used.

3.3.2.1 Status Code

The response MUST contain an HTTP status code as defined in [IETF RFC 2616, 1999].

3.3.2.2 HTTP Headers

HTTP headers MUST only be used to transmit representation metadata and message control information. Parameters for the operation to be performed on the resource MUST NOT be sent as HTTP headers.

It is RECOMMENDED that only standard headers are used. An informational list of these HTTP headers for an IANA registry header fields can be found in [IETF RFC 4229, 2005]. However, well-documented custom headers MAY be used.

The set of RECOMMENDED and REQUIRED response headers are included in Table 5.

Table 5
Response HTTP headers

HTTP header field name	Required	Notes
`Cache-Control`	RECOMMENDED	This is used to specify how all entities will cache the information. For further details on this and other caching headers see Section 3.3.2.4.
`Content-Length`	RECOMMENDED	This specifies the length of the body of the message (if known).
`Content-Type`	REQUIRED	This specifies the format of the body sent in the response. For further information see Section 3.3.1.4.
`Date`	REQUIRED	This specifies the date and time that the HTTP response was originated.
`Location`	OPTIONAL	Used to support redirections in cases where the URI of a resource is not persistent, as described in Section 3.3.1.3.3.

3.3.2.3 Cookies

According to [Fielding, 2000] "Cookies ... violate REST because they allow data to be passed without sufficiently identifying its semantics". Essentially, cookies are used to maintain state, which is in violation of the principles of REST.

However, this SIP recognizes that cookies are widely used to contain session and authentication data. Therefore:

- Cookies SHALL be used in accordance with [IETF 2965, 2000].
- Cookies MAY be used to contain session data.
- Cookies MAY be used to contain authentication data.
- Cookies MAY NOT be used to contain other data.

3.3.2.4 Caching

It is RECOMMENDED that *Web Service Providers* (origin servers, services) support caching.

Web Service Providers that claim support for caching SHALL support caching for the HTTP GET and HEAD verbs only.

A URI path that contains a query element SHOULD be treated as un-cacheable.

Web Service Consumers and *Web Service Providers* on the network SHOULD have synchronized time sources.

3.3.2.4.1 Expiration Caching

Web Service Consumers that claim support for caching SHALL support the HTTP header fields: `Expires` and `Date`; or, `Cache-Control`.

Table 6 specifies the HTTP Response header fields required for expiration caching.

Table 6
Specifications for HTTP response header fields required for expiration caching

HTTP header field name/ (header field value)	Description	Conditions
`Cache-Control/ (max-age)`	REQUIRED header field that specifies the amount of time before the information object expires.	
`Expires`	OPTIONAL header field that specifies the expiration time of the information object.	REQUIRED if the `Cache-Control` header field with a header field value of `max-age` is not set in the HTTP response.
`Date`	REQUIRED header field that specifies the date and time at which the HTTP response was originated.	Used in conjunction with the `Expires` header field to determine the amount of time before the information object expires.

3.3.2.4.2 Validation Caching

Validation caching allows the *Web Service Consumer* to interrogate the *Web Service Provider* for the purposes of determining if the cached HTTP response is still valid.

Web Service Consumers that claim support for validation caching SHALL support the HTTP `Cache-Control` header field with values: `no-cache`; `max-age=0`; `must-revalidate`; and, `proxy-revalidate`.

Web Service Consumers and *Web Service Providers* that claim support for validation caching MAY support the HTTP `ETag` header field, HTTP `If-None-Match` header field and HTTP `If-Match` header field to determine if the cached response has changed.

Web Service Consumers and *Web Service Providers* that claim support for validation caching SHALL support the HTTP `Last-Modified` header field to determine if the cached response has changed.

Table 7 specifies the HTTP response header fields required for validation caching.

Web Service Providers that claim support for validation caching SHALL return a `304` (not modified) HTTP status code to indicate that the response has not been modified.

3.3.2.4.3 Security

Caching of sensitive information by *Web Service Consumers* SHALL be prohibited.

Web Service Consumers SHALL NOT cache information contained within an HTTP response that contains an HTTP `Cache-Control` header field with the values: `no-store`; `no-cache`; and, `private`.

Additional constraints, such as digital encryption and digital signatures, SHOULD be applied in order to provide additional protection for confidentiality, availability and integrity of sensitive information objects.

3.3.3 **Errors**

An error can be encountered either based on the submitted HTTP request from the *Web Service Consumer* or because of issues encountered within the *Web Service Provider*.

An error SHALL be conveyed in accordance with [IETF RFC 2616, 1999] status codes.

It is RECOMMENDED that an HTTP status code of 4xx is returned in the HTTP response when an error occurs due to the HTTP request from the *Web Service Consumer*.

It is RECOMMENDED that an HTTP status code of 5xx is returned in the HTTP response when an error occurs within the *Web Service Provider*.

The RECOMMENDED list of HTTP status codes to be conveyed as 4xx or 5xx errors are provided at [IANA HTTP Status Codes, 2012].

It is RECOMMENDED that the HTTP response, indicating the error, includes a `Date` header field with the date and time stored as the `Date` header field value indicating the time the error occurred.

Table 7
Specifications for HTTP response header fields required for validation caching

HTTP header field name/ (header field value)	Description	Conditions
`Cache-Control/ (max-age=0)`	OPTIONAL header field and header field value that instructs the *Web Service Consumer* to revalidate the HTTP Response.	
`Cache-Control/ (no-cache)`	OPTIONAL header field and header field value that instructs the *Web Service Consumer* to revalidate the HTTP Response as caching is not permitted.	REQUIRED header field and header field value if information object MUST NOT be cached.
`Cache-Control/ (must-revalidate)`	REQUIRED header field and header field value to instruct the *Web Service Consumer* to revalidate the cached information object when the information object has expired.	
`Cache-Control/ (proxy-revalidate)`	OPTIONAL header field and header field value for intermediaries.	
`Cache-Control/ (public)`	OPTIONAL header field and header field value.	RECOMMENDED header field and header field value when the `Cache-Control` header field and `proxy-revalidate` header field value is set in the HTTP Response.
`Last-Modified`	REQUIRED header field indicating the last time the information object was modified.	
`ETag`	OPTIONAL header field representing a unique identifier for the requested resource.	

4 REFERENCES

[Fielding, 2000]:
R. Fielding (on-line), http://www.ics.uci.edu, "Architectural Styles and the Design of Network-based Software Architectures", Ph.D. Thesis, University of California, Irvine, at http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000, viewed 29 August 2012.

[IANA HTTP Status Codes, 2012]:
Internet Assigned Numbers Authority (online), http://www.iana.org, "Hypertext Transfer Protocol (HTTP) Status Code Registry", at http://www.iana.org/assignments/http-status-codes/http-status-codes.xml, 1 May 2012, viewed 29 August 2012.

[IETF RFC 1035, 1987]:
Internet Engineering Task Force (on-line), http://www.ietf.org, Request for Comments 1035, "Domain Names – Implementation and Specification", P.V. Mockapetris, at http://tools.ietf.org/html/rfc1035, November 1987, viewed 29 August 2012.

[IETF RFC 2119, 1997]:
Internet Engineering Task Force (on-line), http://www.ietf.org, Request for Comments 2119, "Key Words for Use in RFCs to Indicate Requirement Levels", S. Bradner, at http://tools.ietf.org/html/rfc2119, March 1997, viewed 29 August 2012.

[IETF RFC 2616, 1999]:
Internet Engineering Task Force (on-line), http://www.ietf.org, Request for Comments 2616, "Hypertext Transfer Protocol -- HTTP/1.1", R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, at http://tools.ietf.org/html/rfc2616, June 1999, viewed 29 August 2012.

[IETF RFC 2817, 2000]:
Internet Engineering Task Force (on-line), http://www.ietf.org, Request for Comments 2817, "Upgrading to TLS within HTTP/1.1", R. Khare, S. Lawrence, at http://tools.ietf.org/html/rfc2817, May 2000, viewed 29 August 2012.

[IETF RFC 2965, 2000]:
Internet Engineering Task Force (on-line), http://www.ietf.org, Request for Comments 2965, "HTTP State Management Mechanism", D. Kristol, L. Montulli, at http://tools.ietf.org/html/rfc2965, October 2000, viewed 29 August 2012.

[IETF RFC 3023, 2001]:
Internet Engineering Task Force (on-line), http://www.ietf.org, Request for Comments 3023, "XML Media Types", M. Murata, S.S. Laurent, D. Kohn, at http://tools.ietf.org/html/rfc3023, January 2001, viewed 29 August 2012.

[IETF RFC 3986, 2005]:
Internet Engineering Task Force (on-line) http://www.ietf.org Request for Comments 3986, "Uniform Resource Identifier (URI): Generic Syntax", T. Berners-Lee, R. Fielding, L. Masinter, at http://tools.ietf.org/html/rfc3986, January 2005, viewed 29 August 2012.

[IETF RFC 4229, 2005]:
Internet Engineering Task Force (on-line), http://www.ietf.org, Request for Comments 4229, "HTTP Header Field Registrations", M. Nottingham, J. Mogul, at http://tools.ietf.org/html/rfc4229, December 2005, viewed 29 August 2012.

[ISO 8601:2004, 2004]:
International Organization for Standardization (on-line), http://www.iso.org, Standard ISO 8601:2004, "Data Elements and Interchange Formats – Information Interchange – Representation of Dates and Times", 3 December 2004, viewed 29 August 2012.

[Leonard and Ruby, 2007]:
R. Leonard, S. Ruby, "RESTful Web Services", O'Reilly, 2007.

[Nottingham, 2005]:
M. Nottingham (on-line), http://tools.ietf.org, "POST Once Exactly (POE)", at http://tools.ietf.org/html/draft-nottingham-http-poe-00, 19 March 200, viewed 29 August 2012.

[Sun Microsystems Laboratories, Inc., 1994]:
Sun Microsystems Laboratories Inc. (on-line), http://labs.oracle.com, Note "A Note on Distributed Computing", J. Waldo, G. Wyant, A. Wollrath, S. Kendall, at http://labs.oracle.com/techrep/1994/smli_tr-94-29.pdf, November 1994, viewed 31 August 2012.

[Tidepedia "C3 Taxonomy", 2012]:
Tidepedia article (on-line), http://tide.act.nato.int, "NATO C3 Classification Taxonomy", at http://tide.act.nato.int/tidepedia/index.php?title=NATO_C3_Classification_Taxonomy, 2012, viewed 31 August 2012.

[W3C RDF-Primer, 2004]:
World Wide Web Consortium (on-line), http://www.w3.org, "RDF Primer", W3C Recommendation, E. Miller, F. Manola, at http://www.w3.org/TR/2004/REC-rdf-primer-20040210/, 10 February 2004, viewed 31 August 2012.

[W3C WSDL 2.0: Primer, 2007]:
World Wide Web Consortium (on-line) http://www.w3.org, "Web Services Description Language (WSDL) Version 2.0 Part 0: Primer", W3C Recommendation, C.K. Liu, D. Booth, at http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626, 26 June 2007, viewed 31 August 2012.

[W3C XLink 1.1, 2010]:
World Wide Web Consortium (on-line), http://www.w3.org, "XML Linking Language (XLink) Version 1.1", W3C Recommendation, E. Maler, D. Orchard, N. Walsh, S. DeRose, at http://www.w3.org/TR/2010/REC-xlink11-20100506/, 6 May 2010, viewed 29 August 2012.

[IETF RFC 2119, 1997]:
Internet Engineering Task Force Request for Comments 2119, "Key Words for Use in RFCs to Indicate Requirement Levels", S. Bradner, IETF, Sterling, Virginia, US, March 1997.

[NC3A RD-2814, 2009]:
NATO Consultation, Command and Control Agency Reference Document 2814, "Bi-SC AIS/NNEC SOA Implementation Guidance" *(provisional title)*, J. Busch, S. Brown, R. Fiske, G. Hallingstad, M. Lehman, NC3A, The Hague, Netherlands, unpublished document dated December 2009 (NATO Unclassified).

[NCIA TR/2012/CPW007253/05, 2012]:
NATO Communications and Information Agency Technical Report 2012/CPW007253/05, "Security Services Service Interface Profile Proposal for Security Token Service", R. Fiske, M. Lehmann, R. Malewicz, L. Schenkels, D. Gujral, NCIA, The Hague, Netherlands, October 2012 (NATO Unclassified).

[NCIA TR/2012/CPW007253/06, 2012]:
NATO Communications and Information Agency Technical Report 2012/CPW007253/06, "Security Services Service Interface Profile Proposal for A Policy Enforcement Point", R. Fiske, M. Lehmann, R. Malewicz, L. Schenkels, D.Gujral, NCIA, The Hague, Netherlands, October 2012 (NATO Unclassified).

[NCIA TR/2012/SPW008000/30, 2012]:
NATO Communications and Information Agency Technical Report 2012/SPW008000/30, "Messaging Service Interface Profile Proposal", R. Fiske, M. Lehmann, NCIA, The Hague, Netherlands, October 2012 (NATO Unclassified).

[OASIS Delegation, 2009]:
Organization for the Advancement of Structured Information Standards (on-line), http://www.oasis-open.org, SAML V2.0 Condition for Delegation Restriction Version 1.0, at http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-delegation.pdf, 15 November 2009, viewed 30 March 2011.

[OASIS SAML, 2005]:
Organization for the Advancement of Structured Information Standards (on-line), http://www.oasis-open.org, Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0., at http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf, 15 March 2005, viewed 30 March 2011.

[OASIS SAML Token Profile, 2006]:
Organization for the Advancement of Structured Information Standards (on-line), http://www.oasis-open.org, Web Services Security: SAML Token Profile 1.1, at http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SAMLTokenProfile.pdf, 1 February 2006, viewed 30 March 2011.

[OASIS WS-Security, 2006]:
Organization for the Advancement of Structured Information Standards (on-line), http://www.oasis-open.org, Web Services Security: SOAP Message Security 1.1, at http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf, 1 February 2006, viewed 30 March 2011.

[OASIS WS-SecurityPolicy, 2009]:
Organization for the Advancement of Structured Information Standards (on-line), http://www.oasis-open.org, WS-SecurityPolicy 1.3, at http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/ws-securitypolicy.html, 2 February 2009, viewed 30 March 2011.

[OASIS WS-Trust, 2009]:
Organization for the Advancement of Structured Information Standards (on-line), http://www.oasis-open.org, "WS-Trust 1.4" at http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.doc, 2 February 2009, viewed 30 March 2011.

[W3C WS-Addressing, 2006]:
World Wide Consortium (on-line), http://www.w3.org, Web Services Addressing 1.0 – Core, at http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/, 9 May 2006, viewed 30 March 2011.

[W3C XML-Encryption, 2002]:
World Wide Consortium (on-line), http://www.w3.org, XML Encryption Syntax and Processing, at http://www.w3.org/TR/xmlenc-core/, 10 December 2002, viewed 30 March 2011.

[W3C XML-Signature, 2002]:
World Wide Consortium (on-line), http://www.w3.org, XML-Signature Syntax and Processing, at http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/Overview.html, 12 February 2002, viewed 30 March 2011.

[WS-Federation, 2006]:
WS-Federation (on-line), Web Services Federation Language (WS Federation) Version 1.1, at http://specs.xmlsoap.org/ws/2006/12/federation/ws-federation.pdf, December 2006, viewed 30 March 2011.

[WS-I Security, 2010]:
Web Services Interoperability Organization (on-line), http://www.ws-i.org, Basic Security Profile Version 1.1, at http://www.ws-i.org/Profiles/BasicSecurityProfile-1.1.html, 24 January 2010, viewed 30 March 2011.

5 ABBREVIATIONS

API	Application programming interface
CIS	Communications and information system
CORBA	Common object request broker architecture
CRUD	Create, read, update and delete
DCE	Distributed computing environment
DCOM	Distributed component object model
DNS	Domain name system
FQDN	Fully qualified domain name
HATEOAS	Hypertext as the engine of application state
HTML	Hypertext markup language
HTTP	Hypertext transfer protocol
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
IP	Internet protocol
JPEG	Joint Photographic Experts Group
JSON	JavaScript object notation
NNEC	NATO Network Enabled Capability
NU	NATO Unclassified
POE	Post once exactly
REST	Representational state transfer
RMI	Remote method invocation
RPC	Remote procedure call
SIP	Service Interface Profile

SOAP	Simple object access protocol
SQL	Structured query language
URI	Uniform resource identifier
WSDL	Web services description language
XML	Extensible markup language