RESTful Architecture Whitepaper for IATA NDC

<table>
<thead>
<tr>
<th>Version</th>
<th>1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Draft</td>
</tr>
<tr>
<td>Document Release Date</td>
<td>[24 Feb 2015]</td>
</tr>
</tbody>
</table>
# Table of Contents

Document Management .......................................................................................................................... 5
Revision History .......................................................................................................................................... 5
References .................................................................................................................................................. 5

1 What is IATA NDC ................................................................................................................................. 6
2 Description of used SOAP methodology .............................................................................................. 6
3 Methodology of ROA Resources Oriented Architecture ........................................................................ 6
   3.1 Overview ........................................................................................................................................ 6
   3.2 Key Principles .................................................................................................................................. 8
   3.3 What different with RESTful Services ............................................................................................. 9
4 Designing Guideline for Resources ..................................................................................................... 9
   4.1 Overview ........................................................................................................................................ 9
   4.2 Rules for defining resource identifiers ............................................................................................ 10
   4.3 Examples ....................................................................................................................................... 11
5 Representation Formats ........................................................................................................................ 12
   5.1 Overview ........................................................................................................................................ 12
   5.2 JSON .............................................................................................................................................. 12
   5.3 XML ............................................................................................................................................... 13
   5.4 Content type ................................................................................................................................... 13
6 Meaning of GET / PUT / POST / DELETE / HEAD ................................................................................ 14
   6.1 GET ................................................................................................................................................ 14
   6.2 POST ............................................................................................................................................. 14
   6.3 PUT ................................................................................................................................................. 15
   6.4 PATCH .......................................................................................................................................... 15
   6.5 DELETE ......................................................................................................................................... 16
   6.6 HEAD ............................................................................................................................................ 16
   6.7 OPTIONS ...................................................................................................................................... 16
7 HATEOAS ............................................................................................................................................. 17
8 REST and ROA (Resource Oriented Architecture) Best Practices ...................................................... 19
   8.1 Abstraction ..................................................................................................................................... 19
   8.2 Usability ......................................................................................................................................... 20
   8.3 Security and control in the system ................................................................................................. 20
   8.4 Rate Limiting .................................................................................................................................. 20
   8.5 Caching .......................................................................................................................................... 20
12.4 Level 3: Hypermedia controls ................................................................. 33
13 Guidance leveraging SOAP assets for RESTful architecture.......................... 33
13.1 RMM Level 0 NDC Example ...................................................................... 33
13.2 RMM Level 1 NDC Example ...................................................................... 46
13.3 RMM Level 2 NDC Example ...................................................................... 51
13.4 RMM Level 3 NDC Example ...................................................................... 52
13.5 General considerations adopting JSON message format .............................. 54
  13.5.1 Use of XML Attributes .......................................................................... 54
  13.5.2 Encoding special characters ................................................................... 55
  13.5.3 Validating JSON documents ................................................................... 56
  13.5.4 Using XML namespaces ........................................................................ 56

Figures

Figure 1 - Google Trends Interest - “SOAP Service” and “REST Service” terms ........... 7
Figure 2 - Number of APIs registered in Programmable Web API directory(http://www.programmableweb.com) ........................................................................... 8
Figure 3 - Meaningful URI hierarchy according to NDC ........................................ 10
Figure 4 - Sequence diagram that shows interactions involved into Web Server Flow ....................................................... 27
Document Management

Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Name</th>
<th>Changes</th>
<th>Version #</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.Dec.2014</td>
<td>HP T&amp;T Practice Senior Architect: Roni Schuetz - <a href="mailto:roni.schuetz@hp.com">roni.schuetz@hp.com</a> HP Senior Architect: Dmytro Khaynatskyy - <a href="mailto:dmytro.khaynatskyy@hp.com">dmytro.khaynatskyy@hp.com</a></td>
<td>Whitepaper content</td>
<td>Version 1.1</td>
</tr>
<tr>
<td>13.Jan.2015</td>
<td>HP T&amp;T Practice Senior Architect: Roni Schuetz - <a href="mailto:roni.schuetz@hp.com">roni.schuetz@hp.com</a> HP Senior Architect: Dmytro Khaynatskyy - <a href="mailto:dmytro.khaynatskyy@hp.com">dmytro.khaynatskyy@hp.com</a></td>
<td>Updated mostly language related feedbacks</td>
<td>Version 1.2</td>
</tr>
<tr>
<td>11.Feb.2015</td>
<td>HP T&amp;T Practice Senior Architect: Roni Schuetz - <a href="mailto:roni.schuetz@hp.com">roni.schuetz@hp.com</a> HP Senior Architect: Dmytro Khaynatskyy - <a href="mailto:dmytro.khaynatskyy@hp.com">dmytro.khaynatskyy@hp.com</a></td>
<td>Rework chapter 12 &amp; 13</td>
<td>Version 1.3</td>
</tr>
<tr>
<td>16.Feb.2015</td>
<td>HP T&amp;T Practice Senior Architect: Roni Schuetz - <a href="mailto:roni.schuetz@hp.com">roni.schuetz@hp.com</a> HP Senior Architect: Dmytro Khaynatskyy - <a href="mailto:dmytro.khaynatskyy@hp.com">dmytro.khaynatskyy@hp.com</a></td>
<td>Updated mostly language related feedbacks</td>
<td>Version 1.4</td>
</tr>
<tr>
<td>24.Feb.2015</td>
<td>HP Senior Architect: Dmytro Khaynatskyy - <a href="mailto:dmytro.khaynatskyy@hp.com">dmytro.khaynatskyy@hp.com</a></td>
<td>Update 13.5.1: Add name conflict issue found in NDC schemas</td>
<td>Version 1.5</td>
</tr>
</tbody>
</table>

References

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>#2</td>
<td>Overview how various major API providers version their APIs:</td>
<td><a href="http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/">http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/</a></td>
</tr>
<tr>
<td>#5</td>
<td>Current list Examples, tools and libraries that support JSON Schema</td>
<td><a href="http://json-schema.org/">http://json-schema.org/</a></td>
</tr>
<tr>
<td>#12</td>
<td>OAuth Community Site</td>
<td><a href="http://oauth.net/">http://oauth.net/</a></td>
</tr>
<tr>
<td>#13</td>
<td>Full tutorial provided by Eran Hammer can be found here</td>
<td><a href="http://hueniverse.com/oauth/guide/intro/">http://hueniverse.com/oauth/guide/intro/</a></td>
</tr>
<tr>
<td>#20</td>
<td>Endpoint Redirection Design Pattern</td>
<td><a href="http://soapatterns.org/design_patterns/endpoint_redirection">http://soapatterns.org/design_patterns/endpoint_redirection</a></td>
</tr>
<tr>
<td>#21</td>
<td>Content Negotiation Design Pattern</td>
<td><a href="http://soapatterns.org/design_patterns/content_negotiation">http://soapatterns.org/design_patterns/content_negotiation</a></td>
</tr>
<tr>
<td>No.</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>#22</td>
<td>Entity Linking Design Pattern</td>
<td><a href="http://soapatterns.org/design_patterns/entity_linking">http://soapatterns.org/design_patterns/entity_linking</a></td>
</tr>
<tr>
<td>#23</td>
<td>Idempotent Capability Design Pattern</td>
<td><a href="http://soapatterns.org/design_patterns/idempotent_capability">http://soapatterns.org/design_patterns/idempotent_capability</a></td>
</tr>
<tr>
<td>#24</td>
<td>Response Caching Design Pattern</td>
<td><a href="http://soapatterns.org/candidate_patterns/response_caching">http://soapatterns.org/candidate_patterns/response_caching</a></td>
</tr>
</tbody>
</table>

### 1 What is IATA NDC

NDC (New Distribution Capability) will enable the travel industry to transform the way air products are retailed to corporations, leisure and business passengers, by addressing the industry’s current distribution limitations: product differentiation and time-to-market, access to full and rich air content and finally, transparent shopping experience.

### 2 Description of used SOAP methodology

The current API is designed by using a set of XML request/response messages without any specific references to the transport protocol, while the messages are created based on business scenarios. The various messages are defined using XML schema definitions language, which allows very detailed specification of all message aspects. To enforce consistent and predictable usage of the XML messages, a number of common types and a set of value constraints are defined in the XML schemas. Messages are defined in a way that stateless communication between client and server is assumed, i.e. all required data to fulfill any given request is present within a single XML message. Within a business context it is safe to assume that only partial data needs to be sent as the requestor / responder already has certain information available to accomplish a business transaction.

While the messages are designed more for an RPC-style protocol like SOAP, the schema does not mandate using SOAP protocol. When these messages are used in the context of the SOAP protocol, they represent the message body of the SOAP message. However it is also possible to partially reuse these message in the context of RESTful architecture.

### 3 Methodology of ROA Resources Oriented Architecture

#### 3.1 Overview

Resource-oriented architecture (ROA) is a style of software architecture and a programming paradigm for designing and developing software in the form of resources with “RESTful” interfaces.

The term **RE**presentational **St**ate **T**ransfer (REST) and the ideas behind it were originally introduced and collated in Dr. Roy Fielding’s thesis, “Architectural Styles and the Design of Network-based Software Architectures” [#1], as an architectural style for building large-scale distributed hypermedia systems. While the ideas defined in the thesis can be applied to any transport protocol, the term REST is mostly used in conjunction with HTTP, and also in this document we always refer to REST over HTTP protocol only.
Over the last years these principles have become increasingly popular and widely adopted by many global enterprises.

As can be seen on these Google Trends comparisons, RESTful services are searched for even more often than SOAP services these days.

![Google Trends Comparison](http://www.google.com/trends/explore#q=SOAP%20Service%2C%20rest%20service)

**Figure 1 - Google Trends Interest - “SOAP Service” and “REST Service” terms**

<table>
<thead>
<tr>
<th>Link to Google Trends</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="http://www.google.com/trends/explore#q=SOAP%20Service%2C%20rest%20service" alt="Link" /></td>
</tr>
</tbody>
</table>

Additionally, the following statistic from [http://www.programmableweb.com](http://www.programmableweb.com), one of the leading sources for information about Web APIs, shows that the number of WEB APIs (mostly these are REST conform to some degree) has grown significantly over the last years.
3.2 Key Principles

The REST architectural style is based on the following basic principles:

**CLIENT-SERVER COMMUNICATION**

Client-server architectures have a very distinct separation of concerns. All applications built in the RESTful style must also be client-server in principle. HTTP is the most popular client-server protocol used to implement RESTful services, therefore in subsequent chapters we will assume and discuss RESTful services over HTTP.

**RESOURCE IDENTIFICATION THROUGH URI**

A RESTful web service exposes a set of resources which identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global address space for resource and service discovery.

**UNIFORM INTERFACE**

All components must interact through a single uniform interface. Because all component interaction occurs via this single interface, interaction with different services is very simple. This also means that the implementation of changes can be made in isolation. Such changes will not affect fundamental component interactions because the uniform interface always remains unchanged. In case of HTTP, this interface is defined by standard HTTP verbs, which are GET, POST, PUT, DELETE, OPTIONS, and HEAD.

**SELF-DESCRIPTIVE MESSAGES**
Resources are decoupled from their representation so that their content can be accessed in a variety of formats (e.g., HTML, XML, plain text, PDF, JPEG, etc.). Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Name</td>
</tr>
<tr>
<td>Message Name</td>
</tr>
</tbody>
</table>

**Stateless**

Every interaction with a resource is stateless, i.e., request messages are self-contained. The server must be able to completely understand the client request without using any server context or server session state.

**Cacheable**

As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not, to prevent clients reusing stale or inappropriate data in response to further requests. Well managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

Any data marked as cacheable may be reused as the response to the same subsequent request. In the case of HTTP, out of the box HTTP features such as ETag or Last-Modified headers are used.

### 3.3 What different with RESTful Services

It is important to note that while SOAP defines a standard for a protocol, REST (REpresentational State Transfer) on the other hand is an architectural style for network based software.

SOAP enforces tight coupling between client and server, i.e. it is highly important that both client and server use exactly the same message format and the client needs prior knowledge of everything it will be using, and otherwise it will not be able to issue the calls. Conversely, in a REST architecture a client is expected to issue REST service calls without much prior knowledge of the API detail. Only a resource identifier (URI) and the media type are required.

RESTful services mostly use XML and JSON to represent its content but are not restricted to those formats and can support any other format like HTML, ATOM, Text, etc.

### 4 Designing Guideline for Resources

#### 4.1 Overview

The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today’s weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In
other words, any concept that might be the target of an author’s hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time. - Roy Fielding’s dissertation.

A resource can be compared to an object in an object-oriented programming language that has type, associated data, and relationships to other resources, but with only a limited unified set of standard methods that operate on it. In case of HTTP these operations are standard HTTP GET, POST, PUT and DELETE methods.

A resource is identified by its resource identifier, typically this is the URI that points to the resource, such as:

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>

Each URI can point to a single resource that either exists globally or is a part of a collection of resources. Each collection is considered to be a resource itself, it always contains resources of the same type and is unordered.

Creating a predictable and meaningful URI hierarchy is very critical to overall understandability and usability of a RESTful API. REST architectural principles do not explicitly define how resources should be defined, but some best practices already exist.

<table>
<thead>
<tr>
<th>Meaningful URI hierarchy according to NDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area Resources</td>
</tr>
<tr>
<td>offers</td>
</tr>
<tr>
<td>offers</td>
</tr>
<tr>
<td>orders</td>
</tr>
<tr>
<td>orders</td>
</tr>
<tr>
<td>orders</td>
</tr>
</tbody>
</table>

* Sub-Resources are optional parameters and only used for further granularity on a given root resource

4.2 Rules for defining resource identifiers

Generally it is better to define many smaller resources then one complex resource with many operations on it. Having one or few larger resources will make the overall architecture more RPC like.
Resource names should always be nouns, and the use of verbs as resource names should be avoided. HTTP methods should be used to specify the verb portion of the request.

Use plurals in URL segments to keep your API URLs consistent across all HTTP methods, and avoid inconsistencies with nouns that have different singular and plural form like person/people.

<table>
<thead>
<tr>
<th>Example</th>
<th>Good</th>
<th>Poor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/flights/XX111/passengers/1111/bags/1</td>
<td>/flight/XX111/passenger/1111/bag/1</td>
</tr>
</tbody>
</table>

When defining collections, plural forms should still be used without extra names such as ‘collection’ or ‘list’:

<table>
<thead>
<tr>
<th>Example</th>
<th>Good</th>
<th>Poor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/flights/1111</td>
<td>/flight_col/1111</td>
</tr>
</tbody>
</table>

When defining resources, the hierarchy of resources should be used to express dependencies between resources, although deeply nested resource structures should be avoided, and instead the use of URL query-string parameters to provide additional parameters to the resource should be considered.

<table>
<thead>
<tr>
<th>Example</th>
<th>Good</th>
<th>Poor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/flights/1111/passengers/?cabin=business&amp;status=checkedin&amp;firstName=Smith</td>
<td>/flights/1111/passengers/business/checkedin/Smith</td>
</tr>
</tbody>
</table>

As some web services are case sensitive and some ignore case, all resources should be named using lower case. Words in the resource should be separated using underscores or hyphens.

### 4.3 Examples

Here are some examples of resource identifier breakdowns:

<table>
<thead>
<tr>
<th>Example</th>
<th>Good</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This order resource is identified by the booking reference ORDER-C98C-4759-A20B</td>
<td><a href="http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B">http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>Good</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>In the case of collections, the following URI represents a collection of passengers within the order ORDER-C98C-4759-A20B</td>
<td><a href="http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B/passengers">http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B/passengers</a></td>
</tr>
</tbody>
</table>
To represent a single traveler from the order ORDER-C98C-4759-A20B the conditional parameter may be added, e.g. Lastname, Firstname, Gender - Mr. John Smith. In the second example, a unique identifier can be used for Mr. John Smith. After all, it will be defined by the implementation how the resource URI can be structured.

URI example 1  
http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B/passengers/smithjohnmr/

URI example 2  

5  Representation Formats

5.1  Overview

RESTful architectural principles do not define a specific format for representing resources. It merely relies on the flexibility provided by the content negotiation features of HTTP protocol to choose between a varieties of MIME document types. Resources are typically represented as either XML, JSON or ATOM, but any other format like PDF, HTML, RDF, etc. are also possible.

This can complicate and hinder the interoperability of a RESTful Web service, as for example, clients expecting JSON data will not be able to parse an XML payload if the API only supports XML.

5.2  JSON

JavaScript Object Notation (JSON) emerged as a standard for the easy exchange of JavaScript object data between systems. JSON strings can be parsed and serialized natively in most modern JavaScript environments. JSON is also easier and less resource intensive to parse than XML, since its structure is much cleaner and is directly mapped to objects in modern programming languages.

For detailed syntax specifications see ECMA-404 Standard “The JSON Data Interchange Format”. [#3]

Recently a draft specification for JSON Schema format definition [#4,#5] has been created, which tries to define a language for describing JSON objects in the same way as XML Schema Definition language is doing this for XML.

Overall JSON syntax is relatively simple and self-explanatory. Here is an example of a JSON document:

```json
{
    "Document": {
        "MessageVersion": 1,
        "TransactionID": "06676931-4cdb-4dff-beca-0ffe6d710a3"
    },
    "Passengers": [
        {
            "FirstName": "MAX",
            "LastName": "MUSTERMANN"
        }
    ],
    "Flight": {
        "Departure": {
            "AirportCode": "ZRH",
            "Date": "2014-12-23T00:00:00"
        }
    }
}
```
This object contains three members: Document, Passengers and Flights where Document itself is an object with scalar attributes. Passengers and Flights represent unordered collections of objects.

5.3 XML
XML is ideal for highly structured information due to the extremely powerful XML Schema Definition (XSD) language, which defines how an XML document must be structured and what validation constraints on document values have to be applied. This information can then be used to validate messages without the need to write any code for it.

However, XML Schema Definition language is also known to be very complex and constructs that are often used in it either cannot always be simply mapped into a programming language of choice or are not readily supported by code generation tools. Parsing and validating XML messages can be also memory and CPU intensive, and especially when used on mobile devices this could be a reason for performance degradation.

5.4 Content type
When exchanging messages using a RESTful API, “Media Types” (also known as either MIME Types or Content Types) available from the HTTP protocol are used to specify the format of exchanged message. The HTTP/1.1 protocol uses an “accept” header to define formats that the client will understand. For example, the following header tells the server that the response can be sent either as XML or as a JSON document:
The "Content Type" header specifies the format in which the message body is sent to the server. For example, the following header tells the server that the incoming message is formatted as a JSON document:

```
Content-type: application/json
```

Anti-pattern: REST APIs commonly use either the "application/json" or the "application/xml" media type in the "Content Type" header of an HTTP/1.1 request or response. This is not fully correct, because what is specified in the content type header is only the message format and not its actual type. While it is correct that the message is formatted using languages such as JSON or XML, its content has additional semantics that require special processing beyond simply parsing the message.

To overcome the anti-pattern problem mentioned above, instead of directly using "application/json" or the "application/xml" MIME types, more specialized content types should be used. For example:

```application/x-iata.ndc.v1+json or application/x-iata.ndc.v1+xml```

Another approach used by some API providers such as VMware’s vSphere Director API goes one step further and declares individual content types for each message. For the NDC example above this could be:

```application/x-iata.ndc.ServiceList.v1+json or application/x-iata.ndc.Order.v1+xml```

Using the latter approach simplifies auto discovery, since clients can use this content type to uniquely associate the message with a given schema definition.

### Meaning of GET / PUT / POST / DELETE / HEAD

We already discussed that resources are the fundamental concept in a RESTful API, and that each resource has its own unique URI. The next important part of REST architecture is the actions that can be executed on those resources. In the case of REST over HTTP, these methods are defined using standard HTTP verbs. Normally, not all resources and collections should implement all the methods and it should be defined at API design time what should be made available.

#### 6.1 GET

GET is used to retrieve a resource or, when executed on a collection, a resource list of corresponding resources. GET is expected to be side effect free, and should not modify a resource on which it was executed. A user should be able to freely execute it several times and get consistent results. GET should ensure proper usage of ETag and/or Last-Modified

**ANTI-PATTERN:** Tunneling everything through GET

#### 6.2 POST

In an ideal REST world this method is used to create new resources and should be executed on URIs that point to collection resources where the new resource will be added.

However, when designing an API there are always situations where an intended action does not directly map to one of the available HTTP verbs. As a workaround in such cases, one could create a
special sub-resource of a resource on which the action should be executed. This could be seen as a
queue to which new messages are added for processing, for example:

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>

The usage of such workarounds should represent rather an exception in the API, otherwise the API
could end up being very RPC-like. If the API contains many such actions it could be a sign that either
resources were poorly chosen or implemented, or the API is better suited to an RPC architecture like SOAP

**ANTI-PATTERN:** Tunneling everything through POST

### 6.3 PUT

This verb is executed on a resource, and is intended to update it. Typically a client will put a message
with the complete new content to replace a resource on the server. PUT operations must also be
idempotent, meaning that it is safe to execute the operation several times with the result not changing
beyond the initial application.

Issuing PUT on a larger resources could mean that a lot of data will be sent over the server connection
just to update few properties. In such situations it is possible to define sub-resources that would
update only those properties, for example:

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
</table>

The disadvantage of such an approach could be that the API will get too complex, with a lot of smaller
resources defined. An alternative to using PUT here would be to use the PATCH verb.

### 6.4 PATCH

As defined in RFC 5789, PATCH is intended to update a resource by providing a set of update
instructions. This is an important difference to PUT, where the sent message contains a new version of
the entity to update, while PATCH only contains instructions on how to update the target version. It is
not explicitly defined by RFC 5789 how this set of instructions should look, and any consistent generic
format would satisfy here.

Two additional specifications exist (RFC 6901 JavaScript Object Notation (JSON) Pointer and RFC 6902
JavaScript Object Notation (JSON) Patch) that specify how such a set of instructions could look. For
example, here is a patch message that could be sent to update the street name for a given passenger:

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
**Example**

<table>
<thead>
<tr>
<th>Content-Type: application/json-patch+json</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
</tr>
<tr>
<td>&quot;op&quot;:&quot;replace&quot;,</td>
</tr>
<tr>
<td>&quot;path&quot;:&quot;/details/address/street&quot;,</td>
</tr>
<tr>
<td>&quot;value&quot;:&quot;Some New Str. 1&quot;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

**ANTI-PATTERN:** PATCH is often incorrectly used to send a subset of the resource data in the same format as one would send a PUT request, just to do a partial update of the resource. For example, in order to update only the street name in the passenger address one could just send this subset of the address resource.

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
</tr>
<tr>
<td>&quot;Street&quot;: &quot;Some Str. 12&quot;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

While this approach would technically work, this is considered incorrect usage of PATCH verb.

### 6.5 DELETE

DELETE verb is used to delete a resource pointed to by a URI. When delete is called on a collection resource, all members of that collection are deleted. A successful response should be “200 OK” if the response includes an entity describing the status, “202 (Accepted)” if the action has not yet been executed, or “204 (No Content)” if the action has been enacted but the response does not include an entity. In cases where the resource is not found, “404 (Not Found)” should be returned.

This operation should also be idempotent, i.e. it should be safe to call DELETE on the same resource several times without any side effects and it should not change the resource beyond what happened during the initial DELETE.

### 6.6 HEAD

HEAD verb is used to retrieve resource metadata without actually retrieving the resource content itself. These metadata could be anything which would also be returned with a GET request. A client can use the HEAD request to check if the resource exists without actually retrieving it, or check if it has been modified since last the get (using ETag header) and when it was modified (using Last-Modified header).

### 6.7 OPTIONS

OPTIONS verb returns the available HTTP methods and other options available on the requested resource. Note that in a browser an OPTIONS request is also used in so called “Preflighted” requests, i.e. cross-site requests that go to another domain than the domain of the resource making the request. A “Preflighted” request first sends an HTTP OPTIONS request in order to check if it is allowed to call the required verb from this domain, and only then issues an actual call if allowed to. Therefore for an
API provider it is important to support this verb when the resource is to be accessed from environments where cross-site HTTP requests can happen, such as in a web browser.

7 HATEOAS

HATEOAS [#6, #7], an abbreviation for Hypermedia as the Engine of Application State, is a constraint of the REST application architecture that distinguishes it from most other network application architectures. The principle is that a client interacts with a network application entirely through hypermedia provided dynamically by application servers. A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia. By contrast, in a service-oriented architecture (SOA), clients and servers interact through a fixed interface shared through documentation or an interface description language (IDL).

The HATEOAS constraint decouples client and server in a way that allows the server functionality to evolve independently.

This principle roughly means that a client interacts with a network application entirely through hypermedia (links) provided dynamically by the endpoint in the resource representation. A truly REST client needs no prior knowledge about how to interact with the endpoint beyond a generic understanding of hypermedia, meaning no out-of-band information such as external API documentation is required.

This principle can be compared to how HTML pages are consumed. When we open a news web page for example and want to navigate to an article of interest, we just click on the news headline (hypermedia) and then are navigated to the article page (“change the state”) From here, again we can click any of the available links and navigate further and so on. Note that in order to use that particular site, it was not necessary to read any site-specific documentation about how to get from the news headline to a specific page, nor did we need to find any article IDs, or construct and type any URLs manually in the browser bar. The same principle would apply to any web site, where all we need to know is how to use links (hypermedia) in order to go from one page to another (“change the state”).

The obvious advantage for the web site owner (API provider) here, is that he can freely change the way how the web site is structured and what URLs are used without affecting web site users in any way.

When transferred to a RESTful architecture, this would mean that when a client requests a resource, the resource representation that is returned contains links to other resources upon which it can execute next actions. Any other navigation, based on any other knowledge such as external documentation is not allowed, for example it is not allowed to pick some IDs from the document manually, construct a URL and then execute next action.

REST architectural principles do not specify exactly how these relations should look, although there is a recommendation that can be followed. In the case of JSON format, our response could look like this:

```
Examples
{
   "link":{
      "rel":"self",
      "url":"http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B"
   }
```


## Examples

```
[
  {
    "rel":"http://ndc.iata.com/schemas/orders/payments",
    "url":"http://ndc.iata.com/orders/payments/ORDER-C98C-4759-A20B"
  },
  {
    "Passengers":{
      "link":{
        "rel":"self",
        "url":"http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B/passengers/PAX1"
      },
      "FirstName":"MAX",
      "LastName":"MUSTERMANN"
    },
    "Flights":{
      "link":{
        "rel":"self",
        "url":"http://ndc.iata.com/operations/schedules/departure/XX1068"
      },
      "Departure":{
        "link":{
          "rel":"self",
          "url":"http://ndc.iata.com/references/airports/ZRH"
        },
        "AirportCode":"ZRH",
        "Date":"2014-12-23T00:00:00",
        "Time":"07:05"
      },
      "Arrival":{
        "link":{
          "rel":"self",
          "url":"http://ndc.iata.com/references/airports/FRA"
        },
        "AirportCode":"FRA",
        "Date":"2014-12-23T00:00:00",
        "Time":"07:05"
      }
    }
  }
]`
```
Examples

```json
{
    "AirportCode":"FRA",
    "Date":"2014-12-30T00:00:00",
    "Time":"08:15"
},
"MarketingCarrier":{
    "link":[
        {
            "rel":"self",
            "url":"http://ndc.iata.com/operations/carriers/XX/flights/1068"
        }
    ],
    "AirlineID":"XX",
    "FlightNumber":"1068"
}
}
```

Strictly speaking, an architecture that claims to be RESTful has to comply with this principle. In reality however, only very few APIs that call themselves RESTful fulfill this principle [#8].

In terms of the RESTful Maturity Model, described below, supporting HATEOAS constraint would be the last level of architectural conformance with REST architectural principles.

Even though the principles described here are not widely adopted by API providers, they still can be partially used in your architecture. For example, to split complex messages such that links to other sub resources could be returned instead of returning the whole large object. Also, when returning collections one could optionally only return links to individual resources instead of the complete resource.

8  REST and ROA (Resource Oriented Architecture) Best Practices

8.1  Abstraction

Resource is a key abstraction in RESTful architecture and it is very critical to have a proper design in place. The basic approach to creating resources, discussed in the previous section, is valid at an abstract level but does not define any solutions on how to deal with real world complex domain requirements as they exist in the airline business, for example.

One of the important decisions that has to be made during API design is whether to define many fine grained resources with CRUD operations or coarse grained resources.

Having many fine grained resources may end up in an API being very “chatty” i.e. require several individual calls from the client to perform a business action. This also implies that the business logic details are being shifted from the server to the client, and would eventually require modifying all the clients when the business process is updated.

Conversely, having coarse grained resources would reduce chattiness and keep business process logic on the server, however at the risk of ending up in an RPC-style service, where there are a lot of methods defined on that resource.
The way to deal with such situations is to define business operation or business process resources, so called “intent” resources that express a business or domain level “state of wanting something” or “state of the process towards the end result”. This process is very well described on the ThoughtWorks “REST API Design - Resource Modeling” blog post [#9].

8.2 Usability
Usability, look and feel and handling of the API are crucial to get wide adoption and acceptance by consumers of the API.

While services must be simple and self-explanatory, it is still good to have documentation, preferably, auto generated from the service metadata.

8.3 Security and control in the system
Because an API can be called from any location over the internet, HTTPS (TLS) should always be used to secure the transport. Communication can happen over public wireless networks, where traffic may not be secured or encrypted, allowing eavesdropping or impersonation if authentication credentials are hijacked. Using HTTPS also simplifies authentication efforts, as one can use simple token based authentication like OAuth or even basic authentication over HTTPS.

Error code HTTP 403 should be returned for unauthorized requests. Optionally, additional information may be provided in the response body.

8.4 Rate Limiting
Several global API providers like GitHub, Twitter, and Vimeo implement rate limiting in their APIs. Rate limiting, also called ‘throttling’, is used to prevent clients from issuing too many requests over some defined amount of time. For example, we can limit API calls to a maximum of 100 per hour for cheaper subscription plans while not applying such limits at all for more expensive ones. Rate limits are typically controlled by a set of custom headers, the following non-standard headers used by the Twitter API are typical and are used in many other APIs:

<table>
<thead>
<tr>
<th>HTTP Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Rate-Limit-Limit</td>
<td>The rate limit ceiling for that given request</td>
</tr>
<tr>
<td>X-Rate-Limit-Remaining</td>
<td>The number of requests left for the 15 minute window</td>
</tr>
<tr>
<td>X-Rate-Limit-Reset</td>
<td>The remaining window before the rate limit resets in UTC epoch seconds</td>
</tr>
</tbody>
</table>

Source: https://dev.twitter.com/rest/public/rate-limiting

8.5 Caching
It should be ensured that the resources are cacheable, and that one of the caching techniques available on the HTTP protocol is supported by using the corresponding headers. There are two approaches available:
- Time based – here the Last-Modified header is used. Clients can include this header with all requests and the API can evaluate it, returning “304 Not Modified” status code with an empty body instead of the full representation of the resource.

- Content based – here the ETag header is used. ETag is generated on the server using a hash function and returned with the initial response. All subsequent requests to the same resource contain this ETag and a server can then again calculate an ETag for the same resource again and thereby determine if the resource has changed or not. If the resource produces the same ETag, “304 Not Modified” status code with an empty body is returned.

### 8.6 Error handling

A successful error handling and error reporting strategy is an essential part of an API design. Unfortunately, unlike SOAP, RESTful architecture does not define any conventions for communicating errors back to the client.

The most common approach, which also conforms to the REST principles, is to use HTTP error codes.

In case of any error situation, the application error is mapped to one of the available HTTP error codes. For example for this request:

<table>
<thead>
<tr>
<th>Example</th>
<th>Retrieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td><a href="http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B">http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B</a></td>
</tr>
</tbody>
</table>

When a booking with the booking reference "ORDER-c98c-4759-a20b" does not exist a HTTP "404 Not Found" is returned.

It is also useful when an error response contains extended data, either as custom HTTP headers such as x-error-description or, more preferably, as a message body. This error message should contain as much detail as possible about the error to simplify the usage of the RESTful API. It is important to return error messages in the same format as any successful response would be expected, for example if the user expected JSON representation then an error message should be also in JSON, when XML then in XML.

In general there are three groups of HTTP return codes:

- **2**\(^\ast\) - For success messages
- **4**\(^\ast\) - For client related issues like an incorrect request, not finding the requested data, missing authentication etc.
- **5**\(^\ast\) - Error codes used for server related issues, like a failed DB connection, network error or any other unexpected error situation

Some of the most often used HTTP codes are [10]:

- **200 OK** - Response to a successful GET, PUT, PATCH or DELETE. Can also be used for a POST that does not result in a creation.
- **201 CREATED** - Response to a POST that results in a creation. Should be combined with a location header pointing to the location of the new resource.
204 **No Content** - Response to a successful request that will not be returning a body (like a DELETE request).

304 **Not Modified** - Used when HTTP caching headers are in play.

400 **Bad Request** - The request is malformed, for example if the body does not parse.

401 **Unauthorized** - No or invalid authentication details are provided. Also useful to trigger an authentication popup if the API is used from a browser.

403 **Forbidden** - When authentication succeeded but the authenticated user does not have access to the resource.

404 **Not Found** - When a non-existent resource is requested.

405 **Method Not Allowed** - When an HTTP method is being requested that is not allowed for the authenticated user.

410 **Gone** - Indicates that the resource at this end point is no longer available. Useful as a blanket response for old API versions.

415 **Unsupported Media Type** – Indicates that incorrect content type was provided as part of the request.

422 **Unprocessable Entity** - Used for validation errors.

429 **Too Many Requests** - When a request is rejected due to rate limiting.

**Anti-pattern**: Errors from the service call always succeed with HTTP “200 OK” and error details are contained either in the message itself or in custom HTTP response headers such as “X-ERROR-CODE:404”, “X-ERROR-MESSAGE: “Flight not found””. This approach that tunnels everything through a “200” breaks various REST architectural principles. It hides the semantics of the operation, imposes understanding the content of the message to process an error thereby breaching the self-contained messages constraint. Additionally, this approach breaks the caching capability of the system, since the browser would not be able to decide if the request can be cached or not since in both cases it is a successful HTTP response

### 8.7 ROA & RESTful - Anti-Patterns

In software engineering, an anti-pattern is a pattern that may be commonly used but is ineffective or counterproductive in practice.

There are a few anti-patterns which are important to keep in mind:

- Tunneling everything through GET
- Tunneling everything through POST
- Ignoring caching
- Ignoring response codes
- Misusing cookies
• Forgetting hypermedia
• Ignoring MIME types
• Breaking self-descriptiveness

8.7.1 Tunneling everything through GET
The following URI received two optional parameters:

• Key: “method” with the value “cancelOrder”
• Key: “id” with the value “ORDER-C98C-4759-A20B”

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tunneling by GET</td>
</tr>
</tbody>
</table>

Actually we retrieve data by the verb “GET” but actually we request the resource to execute the method to “cancelOrder”.

8.7.2 Tunneling everything through POST
This is the same as in the anti-pattern example “Tunneling everything through GET” except that in this case it would use the HTTP POST method. The POST contains a body and is not just a URI, which is how SOAP works and is generally violating any REST principles. However, using this approach can be accepted as an intermediate step while moving from SOAP to RESTful architecture and is considered to be Level 1 in the RESTful maturity model discussed later.

8.7.3 Ignoring caching
As discussed above it is very important to ensure that the application is properly using caching capabilities of HTTP protocol. Not using caching may degrade overall performance and unnecessarily increase network load.

8.7.4 Ignoring response codes
HTTP has a very rich set of application-level status codes for handling different scenarios. The well-known scenarios are 200 “OK”, 404 “Not Found” or 500 “Internal server error”. There are many more available, starting from 1xx up to 5xx.

A RESTful application cannot work only with the codes 200 and 500. In SOAP terms it is acceptable to use the status code 200 and return an xml with the error message, however in REST this is not acceptable. There are particular specific codes for each and every action.

8.7.5 Misusing cookies
A cookie should not be used to send a key some server side session, as this is another REST anti-pattern. If a cookie is used to store some information, such as an authentication token, that server can validate without reliance on session state cookies are RESTful. For authentication it is preferable to use basic authentication over HTTPS from a RESTful HTTP point of view.

8.7.6 Forgetting hypermedia
HTTP sets the verbs which should be used: GET, PUT, POST and DELETE. A client should have to know only one single URL for a resource. All other URLs should be communicated via hypermedia as links within resource representations. Also see chapter [HATEOAS] for more information.
8.7.7 Ignoring MIME types
MIME types define the type of representations of a resource based on the needs. A resource can be presented as XML, JSON or even HTML to be consumed by clients. A client can be a Java, JavaScript or C# application. It can be that a browser presents the response as a PDF or JPEG format but an application can get the same data as XML. Using MIME types provides great flexibility! Ignoring to provide MIME types is bad practice.

8.7.8 Breaking self-descriptiveness
By breaking the constraint of self-descriptive content type (“application/pdf”) the flow is finished and the application cannot continue to another resource. It is almost impossible not to fall into this anti-pattern.

8.8 Versioning
There are two main approaches to versioning (comparison of major APIs [#2]). The pragmatic and widely used approach is to include a version number in the URL. This simplifies the browser’s ability to explore the resources and also prevents accidental caching in browsers across different versions.

<table>
<thead>
<tr>
<th>Example</th>
<th>GET <a href="http://ndc.iata.com/v1/orders/bookings/ORDER-C98C-4759-A20B">http://ndc.iata.com/v1/orders/bookings/ORDER-C98C-4759-A20B</a></th>
</tr>
</thead>
</table>

However the downside of this approach is that URIs representing the resource are not stable across different versions of the API.

The other, academically correct approach, which also conforms with RESTful architecture principles, is to include version number as Accept and Content-Type HTTP headers. However, this approach is rarely used by any major API providers.

<table>
<thead>
<tr>
<th>Example</th>
<th>Content-Type : application/x-iata.ndc.v1+json</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant URI</td>
<td>GET <a href="http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B">http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B</a></td>
</tr>
</tbody>
</table>

Several implementations also use some custom headers such as API-Version to communicate the API version. This is semantically incorrect, since it duplicates the existing HTTP functionality used to define resource representations already provided by Accept and Content-Type HTTP headers. One reason when an API designer may be forced to use this approach is when generic MIME types such as ‘application/json’ are used for content negotiation. This however is also considered to be an anti-pattern.

8.9 Using Query parameters
Query parameters should be used to pass additional information together with the resource URL. Using query parameters can help keep the base URI as simple as possible.

Typical examples where query parameters should be used are:
### Example

<table>
<thead>
<tr>
<th><strong>Paging</strong></th>
<th><code>http://ndc.iata.com/operations/flightstatus/arrivals/ZRH/20150321?page=1&amp;size=10</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Partial loading</strong></td>
<td>Example 1: <code>http://ndc.iata.com/operations/flightstatus/arrivals/ZRH/20150321?fields=carrier,flightnumber</code> &lt;br&gt;Info: result should be limited to data which contains the carrier and the particular flight number &lt;br&gt;Example 2: <code>http://ndc.iata.com/orders/bookings/ORDER-C98C-4759-A20B?fields=total</code> &lt;br&gt;e.g. would return only the total value of this booking (XXXX USD)</td>
</tr>
</tbody>
</table>

---

9  **WS-* for SOAP and what to do with REST?**

For SOAP, there is an additional set of specifications that attempt to solve typical problem areas that trading partners face when communicating over the SOAP protocol. For RESTful services, no such defined counterpart exists [#11].

Instead of WS-Security one would rely on HTTPS protocol for REST services.

One area where a standard approach is being established in the meantime is secure authorization using OAuth protocol, and specifically OAuth 2.0, the next evolution of the OAuth protocol. OAuth 2.0 has the goal to simplify client development with specific flows for web applications, desktop applications, mobile phones and living room devices.

### 10 OAUTH

10.1 Overview

OAuth is an open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications [#12].

> Many luxury cars come with a valet key. It is a special key you give the parking attendant and unlike your regular key, will only allow the car to be driven a short distance while blocking access to the trunk and the onboard cell phone. Regardless of the restrictions the valet key imposes, the idea is very clever. You give someone limited access to your car with a special key, while using another key to unlock everything else.....

> ....In the similar manner OAuth provides a method for users to grant third-party access to their resources without sharing their passwords. It also provides a way to grant limited access (in scope, duration, etc.).

> - Eran Hammer, lead author/developer of OAuth 1 protocol
In this sections we give an overview of the recent OAuth 2 specifications, currently being controversially discussed and even criticized by the original lead author of OAuth 1 protocol[#14], OAuth2 specifications seems to offer a few simplifications and be especially more suitable for enterprise environments.

### 10.2 Terminology

OAuth 2 defines the following roles [#15]:

**CLIENT** An application making protected resource requests on behalf of the resource owner and with its authorization. It can be a server-based, mobile (native) or a desktop application.

**RESOURCE OWNER** An entity capable of granting access to a protected resource. Most of the time, it is an end-user who authorizes a client application to access his account. The application's access to the user's account is limited to the "scope" of the authorization granted (e.g. read or write access).

**AUTHORIZATION SERVER** The server issuing access grants tokens to the client after successfully authenticating the resource owner and obtaining authorization.

**RESOURCE SERVER** The server hosting the protected resources, capable of accepting and responding to protected resource requests. In case of RESTful architecture this is your API that requires authorization. Authorization and resource servers are often combined when the resource provider and authorization provider are the same. The following parameters relevant to security are exchanged during the various authorization flows:

**AUTHORIZATION CODE** Used during Web Server Flow and represents grants given by the user to a client application. This code is later presented to an authorization server to get an access token.

**ACCESS TOKEN** Used by the client and presented to the resource server in order to access its protected resources.

**REFRESH TOKEN** This is not required by the specifications but also often used to request a new Access Token from the authorization server when Access Token expires.

**CLIENT ID** Assigned to a client application during registration at the authorization server.

**CLIENT SECRET** A string assigned to a client application during registration at the authorization server. This string is used only in Web Server Flow, since during other flows the confidentiality of this data cannot be guaranteed and will in any case not provide any additional security.

### 10.3 Authorization flows

OAuth2 specification defines several flows that suit different use cases, and the user is free to choose to implement any of them.

#### 10.3.1 Web Server Flow

This is a flow that is typically used by Web Server Applications developed using server based technology where source code is not publicly visible. It uses an Authorization Code given by the user to retrieve an access token and consume protected resources.
Figure 4 - Sequence diagram that shows interactions involved into Web Server Flow

Here a client application, in this example a Web Application at a check-in desk issues a request to an airline authorization server to grant access to check-in functionality for an application with the client id “checkin_gui”.

Example


A client application user is presented with the authorization prompt, typically describing what application and what resources it is going to access. After successful authorization, the user will be redirected back to the Check-in Application, passing the access code as a query parameter.

Example

| Response             | https://airport.com/checkin?code=ACCESS_CODE |

In the next step, the client web application uses this ACCESS_CODE together with client_id and client_secret (assigned during application registration) to retrieve access and (optionally) renew tokens from the authorization server.

Example

The authorization server validates the information provided and if successful returns the required tokens to the client web application.

The flow then passes this access token either as a query parameter or authorization header to all subsequent requests to a resource server, typically a RESTful API.

10.3.2 User-Agent Flow

This flow is suitable for browser based applications typically written using JavaScript, where the entire code is loaded to the user-agent and hence cannot maintain confidentiality anymore. This flow is also used by mobile and desktop applications that use the browser (embedded or external) for authorization.

The flow is very similar to the web server flow, but because access code confidentiality cannot be guaranteed it is not used at all. The access token is returned directly after the user accepts the authorization prompt. In this flow, the identity of the client application is not verified and solely relies on the preconfigured redirect URI.

Also, refresh tokens are not supported in this flow.

From the same scenario as in the web server flow, the request URI looks as follows:

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>

Note that in this case the response_type parameter is a token, as opposed to code in case of the web server flow.

Here, after the user successfully accepts the authorization request he is redirected to the client application URL with the access token passed.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://airport.com/checkin?token=ACCESS_TOKEN">https://airport.com/checkin?token=ACCESS_TOKEN</a></td>
</tr>
</tbody>
</table>

Note that here the token is not passed using a query parameter as in the case of the web application flow but rather using a URL hash. Using a URL hash here prevents the original client application page from being refreshed, while still allowing JavaScript to react when the hash value is updated.

10.3.3 Username and Password Flow

This flow uses a user name and password to be exchanged for a security token. Typically this flow is used by the clients that are created by the resource provider itself, since it involves a high level of trust between the client application and the resource provider.

This flow is very similar to user agent flow, the only difference being that the user name and password are passed to an authorization server and exchanged for an access token.
From the same scenario as in the user agent flow, the client application simply issues the following POST request:

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
</tr>
</tbody>
</table>

The response to this provides an access token in the same way as shown in the web server flow. Note that here the response_type=password is used.

### 10.3.4 Assertion Flow

A draft specification defines an abstract message flow and processing rules during OAuth 2.0 Assertion Flow [#16]. Additionally, there are two detailed specifications that deal with SAML 2.0 assertions [#17] and JSON Web Token (JWT) assertions [#18].

Typically this flow is used in federated security scenarios where SAML assertion is obtained from a federated identity provider and then passed to an authorization server to obtain an access token. The flow is similar to the web application flow, except that an assertion token is used to obtain an access token from the authorization server instead of an access code.

### 10.3.5 Client Credentials Flow

This flow is used when the client application itself needs to access its own data. Here client credentials are exchanged for an access token to access its resources.

In the check-in scenario described above, the request URL would be:

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>

The access token will be returned in the response in the same way as in the web server flow.

## 11 ROA Design Patterns

Various generic SOA design patterns can directly be applied to RESTful architectures. Some of the existing design patterns even map directly to Resource Oriented architectural principles. Here we will show several design patterns and describe how they are used in the context of resource oriented architecture. For details on these and other design patterns, see the publication "SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST (The Prentice Hall Service Technology Series from Thomas Erl)".

### 11.1 Uniform Contract

The "Uniform Contract" pattern [#19] deals with the problem where a service consumer has to access various services and must always know all the evolving details about those service contracts.
The suggested solution in this design pattern is to standardize a uniform contract across alternative service endpoints that is abstracted from the specific capabilities of individual services.

This design pattern maps directly to Uniform interface REST architectural principle discussed above, where the REST architecture is only using HTTP verbs to express the actions that can be executed on resources.

**11.2 Endpoint Redirection**

The “Endpoint Redirection” pattern [#20] addresses the problem that service consumers may encounter when service owners restructure their services and service endpoints may get moved to other URIs or are removed.

As a solution, this pattern suggests automatically referring service consumers that access the stale endpoint identifier to the current identifier.

In the case of REST over HTTP, this design pattern is also automatically supported by the use of location header and the standard HTTP return codes “301 Moved Permanently” and “307 Temporary Redirect”. When a client requests an action on a stale URI, the service must return one of these return codes and the location header must contain the new URI of the resource. The service client can then reissue the call to a new URI provided in the location header.

It is worth noting that this pattern could also be obsolete when HATEOAS principles are supported by the service provider and also used by service consumers. In this situation, service consumers do not rely on a particular structure but instead always use hypermedia to perform actions on different resources. By doing so, if the service provider decides to change the API structure it will return those new URLs in the resource representation message and the client will automatically use them.

**11.3 Content Negotiation**

The “Content Negotiation” pattern [#21] addresses the problem that deals with different service consumers having different requirements regarding how they would like to consume the service meaning a service provider may need to support various representations that are not compatible with each other. Another problem may arise when a service provider needs to change the representation of the service while still supporting old and new service consumers.

The solution suggested by this pattern is to negotiate the specifics of the service representations at runtime as part of the service invocation.

This pattern directly maps to the use of media types in RESTful architecture as discussed above. The service consumer uses content-type and accept HTTP header to specify to the service which message format is used for a request and which message formats could be accepted as a response.

**11.4 Entity Linking**

The “Entity Linking” pattern [#22] deals with the problem where autonomously designed service entities have to be orchestrated by the service consumer. In such a situation, the service consumers have to hard-code all the logic necessary to link these services in order to implement a certain business process. As a result of this, the service consumer must always keep this logic up to date with the business requirements.
The solution to this problem is that the services inform their consumers about the existence of related entities as part of the consumer's interactions with the services.

This design pattern directly maps to HATEOAS principle described above.

### 11.5 Idempotent Capability

The "Idempotent Capability" pattern [#24] deals with the problem that RESTful architectures, as distributed systems, are prone to various network related failures where request messages may get lost or responses may not be delivered to the service consumer. The service consumer should be able to retry the request without any side effects.

The suggested solution to this problem is, wherever possible, to make sure idempotent service capabilities are used. This means that, by design, a service guaranties that it is free of side effects when the same requests are executed several times. This pattern corresponds to the guidelines discussed above regarding where and how HTTP verbs must be used. By design all HTTP verbs used are idempotent, except POST.

### 11.6 Distributed Response Caching

The "Distributed Response Caching" pattern [#24] addresses a performance problem that can exist in distributed client server systems. A service consumer may need to invoke the same service repeatedly and get the same unchanged messages in response. This may unnecessarily increase the response time of the client response and introduce unnecessary overhead on the server in processing those requests.

The solution provided by this pattern is to introduce the caching of response messages at various locations within the technology architecture that underlies the service consumers and services.

The capability provided by the pattern is available out of the box for RESTful architectures through the underlying HTTP protocol. When HTTP headers like ETag and Last-Modified, as discussed in the Best Practices section above, are properly used by both service consumers and service providers, then HTTP GET requests will be cached at various levels of the technology stack, for example browser and network components such as HTTP proxies between a service consumer and an API provider. Additionally, if the request was not cached on the way and hits the service then the service provider may choose to respond with a “304 Not Modified” reply with an empty response message.

### 11.7 Message-Based State Deferral

The "Message-based State Deferral" pattern [#25] addresses the common requirement to manage a session state between separate service invocations. While having to manage state is a valid requirement, often it introduces additional complexity to the overall service architecture and may require additional components that will increase the overall cost of the solution. Typically, in complex scenarios a session cannot be simply kept in memory, but rather has to be stored either in a separate state database or managed by a dedicated state management service. In both cases though, the architecture will become more complex and costly.

This pattern suggests that instead of having the service implementation handle the session, all session-relevant data must always be transported in the request and response messages. This will eliminate the need for a service implementation to hold and manage the session state, since all the required data is readily available in the request.
This design pattern maps directly to the REST architectural principle which states that all interactions between clients and RESTful API must be stateless, and all the messages must be self-contained.

This pattern may however introduce another problem where, with increased message size, additional bandwidth will be required to always transfer the state. This problem can be reduced by ensuring that HTTP content compression is always used.

12 RESTful Maturity Model
When adopting existing SOAP services to RESTful architectural principles, the RESTful Maturity Model (RMM) [#26] can be used as a guide in transitioning from RPC-Style architecture. The idea behind RMM is that it defines levels to which your architecture fulfills REST architectural principles. The next sections will describe these levels and show how an existing NDC schema could be transformed to comply with those levels.

12.1 Level 0: Swamp of POX
This is the starting point. HTTP is just used as a transport protocol to tunnel all requests and responses to and from the remote system. At this level, typically there is only one entry point (URI) and all requests are sent using the HTTP POST method. All responses come with HTTP “200 OK” status and we can deduce whether the call was successful or not from the message content.

12.2 Level 1: Resources
This is the level where, instead of having a single URI in a system, we use different URIs (resources) that represent different entities of the system.

However at this level we still use POST to send our messages, always get HTTP “200 OK” back and inspect the response message to see the status of the operation.

12.3 Level 2: HTTP verbs
Until Level 2, POST was always being used to send messages to the service. At this level we now start using all available HTTP verbs and try to match them as closely as possible to business actions, such as:

- **GET** /orders/bookings/ORDER-C98C-4759-A20B – to retrieve an order
- **DELETE** /orders/bookings/ORDER-C98C-4759-A20B/passengers/PAX1 – to remove the passenger from the order
- **PUT** /orders/bookings/ORDER-C98C-4759-A20B/passengers/PAX1/bags/{id} – to update the weight of the bag that belongs to a passenger on a requested flight.

Another important difference at this level is that we no longer return our messages with HTTP “200 OK”, but also start using HTTP codes such as HTTP “404 Not found” when GET /orders/bookings/ORDER-C98C-4759-A20B does not find the requested order or HTTP “201 Created” when we successfully add a bag.

As a consequence, at this level we also extensively use query parameters when retrieving data using GET and do not have any *RQ messages that read only requests to retrieve data using GET verb.
Particularly the proper usage of the GET verb provides a number of advantages. By using GET we clearly state that this is a safe operation that does not update any state of the resource, and can be freely executed several times without any side effects, thereby clearly separating this operation from other “unsafe” operations using POST/PUT/DELETE verbs. It also automatically adds caching capability to the system that is available on the HTTP protocol out of the box.

12.4 Level 3: Hypermedia controls
This is the highest level and fully conforms to how REST Architectural principles were defined by Dr. Roy Fielding’s thesis. It requires that the HATEOAS principles described above are fully supported.

13 Guidance leveraging SOAP assets for RESTful architecture
As described in this document, a RESTful type of architecture has several key differences to a traditional SOAP architecture. Depending on the desired level of support of a RESTful type of architecture, according to the RMM model, several changes to the existing NDC messages may be required.

Additionally, when JSON is chosen as a supported message format further modifications to the schema will be needed. XML Schema Definition files provide much more flexibility and complexity than can be directly mapped to JSON format. Support of such constructs may require either complex customization in JSON processing or additional rework of existing NDC schemas to be more “JSON friendly”.

To simplify the processing of JSON messages, it could be considered to support only a subset of functionality that is available via traditional SOAP services and define this separately, using dedicated tools available for JSON like JSON Schema[4,27] or document using tools like IODocs (https://github.com/mashery/iodocs).

In this chapter we will provide several examples of how existing NDC messages could be used at various levels of RESTful adoption according to the RMM model described above. We will demonstrate which issues could arise and suggest possible solutions when different message formats are used. Note however that this is not the complete list of issues that may arise when adapting current NDC schemas to a RESTful architecture.

13.1 RMM Level 0 NDC Example
Existing IATA NDC XML messages can be directly used at this level. They have well defined request and response messages, and all that needs to be done is to use the HTTP POST method to send plain XML messages, instead of creating a SOAP request. No modifications to NDC schema definitions are required. Supporting JSON on this level will however already require modifications to existing schemas.

Here is an example where we request a custom REST resource to return an OrderViewRS for a given booking reference:

<table>
<thead>
<tr>
<th>Example XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /NDCOrderService</td>
</tr>
<tr>
<td>Accept: application/xml</td>
</tr>
<tr>
<td>Content-Type: application/xml</td>
</tr>
</tbody>
</table>
**Example XML**

```xml
<OrderRetrieveRQ xmlns="http://www.iata.org/IATA/EDIST">
  <Document>
    <MessageVersion>1</MessageVersion>
  </Document>
  <Party>
    <Sender>
      <AgentUserSender>
        <AgentUserID>SalesAgent</AgentUserID>
      </AgentUserSender>
    </Sender>
  </Party>
  <Query>
    <Filters>
      <BookingReferences>
        <BookingReference>
          <ID>GG8FIW</ID>
          <AirlineID>XX</AirlineID>
        </BookingReference>
      </BookingReferences>
    </Filters>
  </Query>
</OrderRetrieveRQ>
```

**Example JSON**

```json
POST /NDCOrderService
Accept: application/json
Content-Type: application/json

{
  "OrderRetrieveRQ": {
    "Document": {
      "MessageVersion": "1"
    },
    "Party": {
      "Sender": {
        "Item": {
          "$type": "HP.NDC.Services.WebReference.AgentUserSenderType, HP.NDC.Services.WebReference",
          "AgentUserID": {
            "Value": "SalesAgent"
          }
        }
      },
      "Query": {
        "Filters": {
          "BookingReferences": [
            {
              "ID": "GG8FIW",
              "Item": {
                "Value": "XX"
              }
            }
          ]
        }
      }
    }
  }
}
When the Booking reference was found in the system, we get HTTP “200 OK” status and the response is an OrderViewRS message (note that here only a subset of response message is shown):

```
Example JSON
{
  
}
```

```
Example XML
HTTP 200 OK
Content-Type: application/xml

<OrderViewRS xmlns="http://www.iata.org/IATA/EDIST">
  <Document>
    <MessageVersion>1</MessageVersion>
  </Document>
  <Response>
    <OrderViewProcessing>
      <Status>Complete</Status>
    </OrderViewProcessing>
    <Passengers>
      <Passenger ObjectKey="PAX1">
        <PTC>ADT</PTC>
        <Name>
          <Surname>MUSTERMANN</Surname>
          <Given>MAX</Given>
          <Title>MR</Title>
        </Name>
        <Gender>Male</Gender>
      </Passenger>
      <Passengers>
        <Order>
          <OrderID Owner="XX">BFQI2H</OrderID>
          <TotalOrderPrice>
            <Total Code="NOK">175353.0</Total>
          </TotalOrderPrice>
          <OrderItems>
            <OrderItem>
              <OrderItemID Owner="XX">BFQI2H</OrderItemID>
              <FlightItem>
                <Price>
                  <BaseAmount Code="NOK">170660.0</BaseAmount>
                  <Taxes>
                    <Breakdown>
                      <Tax>
                        <Amount Code="NOK">110.0</Amount>
                        <TaxCode>ZN</TaxCode>
                      </Tax>
                      <Tax>
                        <Amount Code="NOK">328.0</Amount>
                        <TaxCode>CH</TaxCode>
                      </Tax>
                      <Tax>
                        <Amount Code="NOK">47.0</Amount>
                        <TaxCode>OO</TaxCode>
                      </Tax>
                      <Tax>
                        <Amount Code="NOK">36.0</Amount>
                        <TaxCode>OP</TaxCode>
                      </Tax>
                    </Breakdown>
                  </Taxes>
                </Price>
              </FlightItem>
            </OrderItem>
          </OrderItems>
        </Order>
      </Passengers>
    </Order>
  </Response>
</OrderViewRS>
```
Example XML

```xml
</Tax>
<Tax>
  <Amount Code="NOK">116.0</Amount>
  <TaxCode>SG</TaxCode>
</Tax>
<Tax>
  <Amount Code="NOK">4056.0</Amount>
  <TaxCode>YQ</TaxCode>
</Tax>
</Breakdown>
</Taxes>
</Price>
</OriginDestination>
<Flight>
  <Indicators PricingIndicator="true"/>
  <Departure>
    <AirportCode>OSL</AirportCode>
    <Date>2015-02-26</Date>
    <Time>19:40</Time>
  </Departure>
  <Arrival>
    <AirportCode>ZRH</AirportCode>
    <Date>2015-02-26</Date>
    <Time>22:00</Time>
  </Arrival>
  <MarketingCarrier>
    <AirlineID>XX</AirlineID>
    <FlightNumber>1217</FlightNumber>
  </MarketingCarrier>
  <CabinType>
    <Code>C</Code>
  </CabinType>
  <ClassOfService>
    <Code>C</Code>
  </ClassOfService>
  <Details>
    <FlightSegmentType>
      <Code>HK</Code>
    </FlightSegmentType>
  </Details>
</Flight>
</OriginDestination>
</FlightItem>
</OrderItem>
</OrderItems>
</Order>
</TicketDocInfos>
<TicketDocInfo>
  <TicketDocument>
    <TicketDocNbr>0000106456518</TicketDocNbr>
    <Type>
      <Code>T</Code>
    </Type>
    <DateOfIssue>2015-02-26</DateOfIssue>
    <CouponInfo>
      <CouponNumber>1</CouponNumber>
      <CouponReference>1</CouponReference>
      <FareBasisCode>
        <Code>CIF</Code>
      </FareBasisCode>
    </CouponInfo>
  </TicketDocument>
</TicketDocInfo>
```
### Example XML

```xml
Example XML

```

### The same response in JSON is:

```json
Example JSON

```

HTTP 200 OK

```json
{
  "OrderViewRS": {
    "Document": {
      "MessageVersion": "1"
    },
    "Item": {
      "$type": "OrderViewRSResponse, http://www.iata.org/IATA/EDIST",
      "OrderViewProcessing": {
        "Status": "Complete"
      },
      "Passengers": {
        "Passenger": [{
          "Gender": {},
          "Name": {
            "Surname": {
              "Value": "MUSTERMANN"
            },
            "Given": [
```

```
Example JSON

```
{
    "Value": "MAX",
},
"Title": "MR",
"PTC": {
    "Value": "ADT"
},
"ObjectKey": "PAX1"
}

"Order": {
    "OrderItems": {
    "OrderItem": [
        "OrderItemID": {
            "Owner": "XX",
            "Value": "BFQI2H"
        },
        "FlightItem": {
            "Price": {
                "BaseAmount": {
                    "Code": "NOK",
                    "Value": 170660.0
                },
                "Taxes": {
                    "Breakdown": {
                        "Tax": [
                            "Amount": {
                                "Code": "NOK",
                                "Value": 110.0
                            },
                            "TaxCode": "ZN"
                        ],
                        "Amount": {
                            "Code": "NOK",
                            "Value": 328.0
                        },
                        "TaxCode": "CH"
                    },
                    "Amount": {
                        "Code": "NOK",
                        "Value": 47.0
                    },
                    "TaxCode": "OO"
                },
                "TaxCode": "OP"
            }
        }
    ]
}
```

### Example JSON

```json
{
    "Code": "NOK",
    "Value": 116.0
},
{
    "TaxCode": "SG"
},
{
    "Amount": {
        "Code": "NOK",
        "Value": 4056.0
    },
    "TaxCode": "YQ"
}
```

`"OriginDestination": [
{
    "Flight": [
        {
            "Indicators": {
                "PricingIndicator": true,
                "PricingIndicatorSpecified": true
            },
            "Departure": {
                "AirportCode": {
                    "Value": "OSL"
                },
                "Date": "2015-02-26T00:00:00",
                "Time": "19:40"
            },
            "Arrival": {
                "AirportCode": {
                    "Value": "ZRH"
                },
                "Date": "2015-02-26T00:00:00",
                "DateSpecified": true,
                "Time": "22:00"
            },
            "MarketingCarrier": {
                "AirlineID": {
                    "Value": "XX"
                },
                "FlightNumber": "1217"
            },
            "CabinType": {
                "Code": "C"
            },
            "ClassOfService": {
                "Code": {
                    "Value": "C"
                }
            },
            "Details": {
                "FlightSegmentType": {
                    "Code": "HK"
                }
            }
        }
    ]
]`
```
Example JSON

```json
{
  "OrderID": {
    "Owner": "XX",
    "Value": "BFQI2H"
  },
  "TotalOrderPrice": {
    "Item": {
      "$type": "CurrencyAmountOptType, http://www.iata.org/IATA/EDIST",
      "Code": "NOK",
      "Value": 175353.0
    }
  },
  "TicketDocInfos": [
    {
      "TicketDocument": {
        "TicketDocNbr": "0000106456518",
        "Type": {
          "Code": "T"
        },
        "DateOfIssue": "2015-02-26T00:00:00",
        "CouponInfo": [
          {
            "CouponNumber": "1",
            "CouponReference": "1",
            "FareBasisCode": {
              "Code": "CIF"
            },
            "CouponMedia": "Electronic",
            "Status": {
              "Code": "O"
            },
            "SoldAirlineInfo": {
              "DepartureDateTime": {},
              "Departure": {
                "AirportCode": {
                  "Value": "OSL"
                },
                "Date": "2015-02-26T00:00:00"
              },
              "Arrival": {
                "AirportCode": {
                  "Value": "ZRH"
                },
                "Date": "2015-02-26T00:00:00",
                "DateSpecified": true
              },
              "MarketingCarrier": {
                "AirlineID": {
                  "Value": "XX"
                },
                "FlightNumber": "1217"
              }
            }
          }
        ]
      }
    }
  ]
}
```
In case of any errors, such as the requested booking reference not being found, we will still get an HTTP "200 OK" with the same response message, but with some error content such as:

**Example XML**

HTTP 200 OK

```xml
<OrderViewRS xmlns="http://www.iata.org/IATA/EDIST">
  <Document>
    <MessageVersion>1</MessageVersion>
  </Document>
  <Errors>
    <Error>
      <Type>13</Type>
      <Code>APP_ERR</Code>
      <Description>DisplayBooking: 10054 Exception Caught While Retrieving Booking: (1) NO MATCH FOUND</Description>
    </Error>
  </Errors>
</OrderViewRS>
```

**Example JSON**

HTTP 200 OK

```json
{
  "OrderViewRS": {
    "Document": {
      "MessageVersion": "1"
    },
    "Item": {
      "$type": "ProcessingErrorException, http://www.iata.org/IATA/EDIST",
      "Error": [
```
As can be seen, for now this does not differ much from the pure RPC-style services such as SOAP.

Note here the usage of Accept and Content-Type headers. At this stage, it is already possible to use different representation formats and send and receive JSON messages as well as XML. XML messages will look exactly the same as inside SOAP requests.

Several problems exist however when we serialize our object to JSON using JSON.NET as in our examples. Similar problems also exist when other programming languages or libraries are used.

Note here the use of the special "$type" field in the JSON request message, since this attribute is not available in either the XML message nor in the XSD.

"$type": "HP.NDC.Services.WebReference.AgentUserSenderType, HP.NDC.Services.WebReference"

This field is required by the JSON serializer in order to know the type of the corresponding object, since in the XSD it is defined as:

```xml
<xsd:element name="Sender">
  <xsd:annotation>
    <xsd:documentation source="description" xml:lang="en">Message Sender information.</xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:choice>
      <xsd:element ref="AgentUserSender" />
      <xsd:element ref="AggregatorSender" />
      <xsd:element ref="EnabledSystemSender" />
      <xsd:element ref="MarketingCarrierSender" />
      <xsd:element ref="ORA_Sender" />
      <xsd:element ref="OperatingCarrierSender" />
      <xsd:element ref="POA_Sender" />
      <xsd:element ref="RetailPartnerSender" />
      <xsd:element ref="TravelAgencySender" />
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

This declaration signifies that “Sender” can contain one of the different “Sender” types. As a result, the JSON serializer has to have the ability to know what exact type was serialized in the Item field. By default, the JSON.NET library will put an extra "$type" field that contains a qualified .NET class name of the serialized object. This information is then used during deserialization to construct the proper type.
As can be seen from the content of the "$type" field, it is implementation specific:

"HP.NDC.Services.WebReference.AgentUserSenderType, HP.NDC.Services.WebReference"

Here it contains a company name in both the class name and the assembly name. Consequently, the consumer of this message will not be able to deserialize the message without also having the .NET libraries used during serialization. Even if one could agree on a common namespace and assembly name, this approach will still not work between different programming languages. In Java there is no concept of assembly names and this would therefore require an extra mapping layer from .Net names to Java class names.

An alternative approach would be to add custom mapping logic and, instead of a qualified class name that is dependent on the programming language, use Type names defined in XSDs. In this case the request would be as follows:

**Example JSON**

```json
POST /NDCOrderService
Accept: application/json
Content-Type: application/json

{
  "OrderRetrieveRQ": {
    "Document": {
      "MessageVersion": "1.4",
      "CreateTimeSpecified": false
    },
    "Party": {
      "Sender": {
        "$type": "AgentUserSenderType, http://www.iata.org/IATA/EDIST",
        "AgentUserID": {
          "Value": "SalesAgent"
        }
      }
    },
    "Query": {
      "Filters": {
        "BookingReferences": [
          {
            "ID": "GG8FIW",
            "Item": {
              "$type": "BookingReferenceTypeAirlineID, http://www.iata.org/IATA/EDIST",
              "Value": "LX"
            }
          }
        ]
      }
    }
  }
}
```

Here "$type" contains "AgentUserSenderType, http://www.iata.org/IATA/EDIST" which can be uniquely mapped to a complex type "AgentUserSenderType" from the XSD.
In some cases however, anonymous type declarations are used in current NDC XSDs. For example, in the case of the BookingReferenceType, there is a choice of two elements defined by anonymous types, namely “AirlineID” and “OtherID”:

```xml
<xsd:complexType name="BookingReferenceType">
    <xsd:complexContent>
        <xsd:extension base="KeyWithMetaObjectBaseType">
            <xsd:sequence>
                <xsd:choice>
                    <xsd:element name="AirlineID">
                        <xsd:complexType>
                            <xsd:complexContent>
                                <xsd:extension base="AirlineID_Type">
                                    <xsd:attribute name="Name" type="ProperNameSimpleType" use="optional"/>
                                </xsd:extension>
                            </xsd:complexContent>
                        </xsd:complexType>
                    </xsd:element>
                    <xsd:element name="OtherID">
                        <xsd:complexType>
                            <xsd:simpleContent>
                                <xsd:extension base="UniqueStringID_SimpleType">
                                    <xsd:attribute name="refs" type="InstanceClassRefSimpleType" use="optional"/>
                                    <xsd:attribute name="ObjectMetaReferences" type="InstanceClassRefSimpleType" use="optional"/>
                                    <xsd:attribute name="Name" type="ProperNameSimpleType" use="optional"/>
                                </xsd:extension>
                            </xsd:simpleContent>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:choice>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

During code generation, custom classes that represent these two elements “BookingReferenceTypeAirlineID” and “BookingReferenceTypeOtherID” are generated. These class names are implementation specific and different code generators may choose to use different approaches to creating such names.

Another problem related to the use of custom attributes that contain type information is that it is not standardized. Every serialization library may choose to use different attribute names for this purpose, which will further complicate interoperability and may require complex customization or preprocessing of messages to convert them to the expected format for a given library. In all the examples above, JSON.NET library was used for serialization, which is using "$type" as an attribute name. Another popular library in the Java world, Jersey, uses "@type" attribute for the same purpose.

---

Possible XSD schema modification

Do not use anonymous types in xsd:choice declaration as shown above
Due to all the problems shown above, we recommend avoiding the use of such additional type information in JSON messages.

This can be achieved by not using "xsd:choice" definitions in XSD messages at all. An alternative to "xsd:choice" could be the use of an "xsd:sequence" of elements, where each element can occur zero or one time. While using "xsd:sequence" in this way is not exactly the same as "xsd:choice", it does solve the problem of using additional type information in custom attributes.

For example, if we change the “Sender” XSD type definition to use “xsd:sequence” as follows:

```
<xsd:element name="Sender">
   <xsd:documentation source="description" xml:lang="en">Message Sender information.</xsd:documentation>
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="AgentUserSender" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="AggregatorSender" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="EnabledSystemSender" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="MarketingCarrierSender" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="ORA_Sender" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="OperatingCarrierSender" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="POA_Sender" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="RetailPartnerSender" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="TravelAgencySender" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

The request would be as follows:

```
Example JSON

POST /NDCOrderService
Accept: application/json
Content-Type: application/json

{
   "OrderRetrieveRQ": {
      "Document": {
         "MessageVersion": "1"
      },
      "Party": {
         "Sender": {
            "AgentUserSender": {
               "AgentUserID": {
                  "Value": "SalesAgent"
               }
            }
         }
      }
   }
}
```

Now “AgentUserSender” is known to always contain objects of the type “AgentUserSenderIdType” and as a result of this the serializer does not need any additional type information.
This example demonstrates that not all features available in XSD can be directly mapped to JSON. Also, if JSON is supposed to be used as a supported message format, then either XSDs must be simplified to contain only a subset supported by JSON or complex customizations and pre- or post-processing of messages may be required to handle JSON messages.

Overall, the JSON format to which objects are serialized is not directly derived from the XSD messages. Instead, a generated code structure is used for JSON serialization. This means that even though the source code for different programming languages was generated from the same XSD, it can be that serializing the same XML in JSON using different programming languages will produce different JSON messages.

### 13.2 RMM Level 1 NDC Example

To support RESTful architectural principles at level 1 according to the RESTful maturity model, further structural changes will be required for all Request messages. Depending on how granular the access to an individual object is required to be, the existing response side messages will need to be split to allow access to individual elements like Passengers, Payments, TicketDocInfos etc. Additionally, further specifications for resource URLs will be required that go beyond NDC message format specifications.

The first step to adopt to this level is to define what resource URLs can be used in the system to represent the different functionality currently provided by existing NDC request response messages. To help identify and build those resource URLs, the following steps could be used:

- **Identify the root level structure**
  At the root level, resources should be grouped by their area, which directly map to the BRDs created for NDC:
  - Offers
  - Orders
  - Fulfillments
  - Operations
  - References

- **Identify the object taxonomy**
  Based on the existing schemas, define all the objects that should be represented by a URL. Some, but not all, such objects are:
  - Airline, Order, Flight, Fare, Airport, Passenger, Ticket, Bag, etc.
  These objects will represent the root resource name portion of the path.

  Each resource can be compared to an object in object oriented programming, and each message that we post to a method on that object.

- **Identify conditional parameters**
  Further, for each such object that needs to be accessible via a URL, conditional parameters should be defined that are used to identify the object. For "Airline" this will be "Airline Designator", for "Flight", this will be "departure date" and "flight number" etc.
These conditional parameters will identify the given resource, will be mandatory and will be used to create the resource path.

- **Identify dependencies between objects**
  To further structure the objects and represent dependencies between them, sub-resources could be also added, for example to specify a URL for passengers belonging to a flight (orders/<ORDER-ID>/passengers) or bags that belong to a passenger (orders/<ORDER-ID>/passengers/PAX1/bags).

- **Define optional parameters**
  All other parameters that may be required to retrieve the resource are optional, and will be specified as URL query parameters. This could be the language in which to return descriptions, paging details when the requested resource is a list etc.

At level 1 however, this is not yet required since we always send a request message that may still contain all the other parameters that are not essential for identifying object via URL.

To help structuring resources a table as below can be used:

<table>
<thead>
<tr>
<th>Name</th>
<th>Area</th>
<th>Root resource name</th>
<th>Version</th>
<th>Identifier</th>
<th>Sub resource name</th>
<th>Identifier</th>
<th>Filters / options</th>
<th>Resource URL structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>City list / info</td>
<td>references</td>
<td>cities</td>
<td>v1</td>
<td><img src="value" alt="cityCode" /></td>
<td></td>
<td><img src="value" alt="languageCode" /></td>
<td></td>
<td><img src="value" alt="api someairline com/v1/references/cities[/cityCode]?lang=languageCode" /></td>
</tr>
<tr>
<td>Airport list / info</td>
<td>references</td>
<td>airports</td>
<td>v1</td>
<td><img src="value" alt="airportCode" /></td>
<td></td>
<td><img src="value" alt="languageCode" /></td>
<td></td>
<td><img src="value" alt="api someairline com/v1/references/airports[/airportCode]?lang=languageCode" /></td>
</tr>
<tr>
<td>Nearest Airport</td>
<td>references</td>
<td>airports</td>
<td>v1</td>
<td>nearest <img src="value" alt="latitude" />, <img src="value" alt="longitude" /></td>
<td></td>
<td><img src="value" alt="languageCode" /></td>
<td></td>
<td><img src="value" alt="api someairline com/v1/references/airports/nearest[/latitude],[longitude]?lang=languageCode" /></td>
</tr>
<tr>
<td>Country list / info</td>
<td>references</td>
<td>countries</td>
<td>v1</td>
<td><img src="value" alt="countryCode" /></td>
<td></td>
<td><img src="value" alt="languageCode" /></td>
<td></td>
<td><img src="value" alt="api someairline com/v1/references/countries[/countryCode]?lang=languageCode" /></td>
</tr>
<tr>
<td>Airlines list / info</td>
<td>references</td>
<td>airlines</td>
<td>v1</td>
<td><img src="value" alt="carrierCode" /></td>
<td></td>
<td></td>
<td></td>
<td><img src="value" alt="api someairline com/v1/references/airlines/carrierCode" /></td>
</tr>
<tr>
<td>Aircraft list / info</td>
<td>references</td>
<td>aircraft</td>
<td>v1</td>
<td><img src="value" alt="aircraftCode" /></td>
<td></td>
<td></td>
<td></td>
<td><img src="value" alt="api someairline com/v1/references/aircraft/aircraftCode" /></td>
</tr>
<tr>
<td>Flight Status by flight</td>
<td>operations</td>
<td>flightstatus</td>
<td>v1</td>
<td><img src="value" alt="flightNumber" />, <img src="value" alt="departureDate" /></td>
<td></td>
<td></td>
<td></td>
<td><img src="value" alt="api someairline com/v1/operations/flightstatus[/flightNumber]?departureDate" /></td>
</tr>
<tr>
<td>Flight Arrivals</td>
<td>operations</td>
<td>flightstatus</td>
<td>v1</td>
<td>arrivals <img src="value" alt="airportCode" />, <img src="value" alt="startDateTime" />, <img src="value" alt="endDateTime" /></td>
<td></td>
<td></td>
<td></td>
<td><img src="value" alt="api someairline com/v1/operations/flightstatus/arrivals[/airportCode][?startDateTime]?endDateTime" /></td>
</tr>
<tr>
<td>Flight Departures</td>
<td>operations</td>
<td>flightstatus</td>
<td>v1</td>
<td>departures <img src="value" alt="airportCode" />, <img src="value" alt="startDateTime" />, <img src="value" alt="endDateTime" /></td>
<td></td>
<td></td>
<td></td>
<td><img src="value" alt="api someairline com/v1/operations/flightstatus/departures[/airportCode][?startDateTime]?endDateTime" /></td>
</tr>
<tr>
<td>City pair</td>
<td>operations</td>
<td>flightstatus</td>
<td>v1</td>
<td>citypair <img src="value" alt="originAirportCode" />, <img src="value" alt="destinationAirportCode" /></td>
<td></td>
<td></td>
<td></td>
<td><img src="value" alt="api someairline com/v1/operations/flightstatus/citypair[/originAirportCode][?originAirportCode][?destinationAirportCode]?destinationAirportCode" /></td>
</tr>
</tbody>
</table>
Now the previous example can be transformed to the following call to a new URL:

```
Example XML

POST /v1/orders/orders?BookingReference=GG8FIW&AirlineID=XX

Accept: application/xml
Content-Type: application/xml

<OrderRetrieveRQ xmlns="http://www.iata.org/IATA/EDIST">
  <PointOfSale>
    <Location>
      <CityCode>BOM</CityCode>
    </Location>
  </PointOfSale>
  <Party>
    <Sender>
      <AgentUserSender>
        <AgentUserID>SalesAgent</AgentUserID>
      </AgentUserSender>
    </Sender>
  </Party>
  </OrderRetrieveRQ>

Note that here we do not need to use a "Document" tag that contains the message version since it is now part of the resource URL:

<Document>
  <MessageVersion>1</MessageVersion>
</Document>

The same also applies to passing the booking reference, instead of a "Query" tag in the XML message:

<Query>
  <Filters>
    <BookingReferences>
      <BookingReference>
        <ID>GG8FIW</ID>
        <AirlineID>XX</AirlineID>
      </BookingReference>
    </BookingReferences>
  </Filters>
</Query>
```
It is simply passed on the URL:

```
POST /v1/orders/bookings?BookingReference=GG8FIW&AirlineID=XX
```

Using a similar approach, other query options can be transformed to URLs as well. For example, instead of the following query tag:

```
<Query>
  <Filters>
    <Passengers>
      <FQTV>
        <AirlineID>XX</AirlineID>
        <Account>
          <Number>99222802657346</Number>
        </Account>
      </FQTV>
    </Passengers>
  </Filters>
</Query>
```

We just use the following:

```
POST /v1/orders/bookings?AirlineID=XX&FQTVNumber=99222802657346
```

Alternatively, to request using the Order id, the URL would be as follows:

```
POST /v1/orders/bookings/ORDER-c98c-4759-a20b
```

Overall, at Level1 we deal primarily with different URLs and simplify the request messages, while the response messages remain largely untouched if we do not provide fine granular access to the individual parts of the original OrderViewRS message.

In this case, we get the same response as in the Level 0 example:

```xml
<Example XML>
HTTP 200 OK

<OrderViewRS xmlns="http://www.iata.org/IATA/EDIST">
  <Response>
    <Order>
      <OrderID Owner="XX">CYEG3M</OrderID>
      <TotalOrderPrice>
        <Total Code="USD">5708.8</Total>
      </TotalOrderPrice>
      <OrderItems>
        <OrderItem>
          ....
        </OrderItem>
      </OrderItems>
    </Order>
  </Response>
</OrderViewRS>
</Example XML>
```

If a finer granular access to the initial NDC messages is already required at this level, we need to ensure that the NDC schema definitions also contain "xsd:element" declarations for all the types that might represent a resource.
In the following example, a sub-resource URL is used to return a list of passengers for a given flight:

```
Example XML

POST /v1/orders/bookings/ORDER-c98c-4759-a20b/passengers
HTTP 200 OK

<Passengers xmlns="http://www.iata.org/IATA/EDIST">
  <Passenger ObjectKey="PAXRPH_1">
    <PTC>ADT</PTC>
    <Name>
      <Surname>TEST</Surname>
      <Given>BERND</Given>
      <Title>MR</Title>
    </Name>
    <FQTVs>
      <TravelerFQTV_Information>
        <AirlineID>LX</AirlineID>
        <Account>
          <Number>333035119064941</Number>
        </Account>
      </TravelerFQTV_Information>
    </FQTVs>
    <Gender>Male</Gender>
  </Passenger>
  <Passenger ObjectKey="PAXRPH_1">
    <PTC>ADT</PTC>
    <Name>
      <Surname>TEST</Surname>
      <Given>FRANC</Given>
      <Title>MR</Title>
    </Name>
    <FQTVs>
      <TravelerFQTV_Information>
        <AirlineID>LX</AirlineID>
        <Account>
          <Number>333035111474978</Number>
        </Account>
      </TravelerFQTV_Information>
    </FQTVs>
    <Gender>Male</Gender>
  </Passenger>
</Passengers>
```

To support this resource message we need to add an additional declaration to the NDC XSD:

```
<xsd:element name="Passengers">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Passenger" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Similar declarations need to be done for all elements that are to be exposed via individual messages.
Possible XSD schema modification

Make sure "xsd:element" definitions exist at root level for all elements that may be exposed as individual resources

13.3 RMM Level 2 NDC Example

When adapting NDC messages to RMM Level 2, it is necessary to completely eliminate *RQ messages for all read-only requests that will use HTTP GET. Instead, all the information from those messages is mapped to URL parameters. Considering the complexity of current RQ messages, it may not be possible or reasonable to map all request parameters from current NDC request messages. In this case, a RESTful API might be defined to only support a subset of the functionality that is available in SOAP.

Now our previous order retrieve examples would be transformed to just a single URL GET request with several query parameters:

**Example URL**

GET /orders/bookings/ORDER-c98c-4759-a20b?currency=CHF
Swiss Francs: GET /orders/bookings/ORDER-C98C-4759-A20B?currency=CHF
Swiss Francs: GET /orders/bookings/ORDER-c98c-4759-a20b?currency=CHF
GET /orders/bookings/ORDER-c98c-4759-A20B/segments

As a response we receive:

**Example XML**

HTTP 200 OK
<Order xmlns="http://www.iata.org/IATA/EDIST">
  <OrderID Owner="XX">CYEG3M</OrderID>
  <TotalOrderPrice>
    <Total Code="CHF">5708.8</Total>
  </TotalOrderPrice>
</Order>

In addition to the proper use of HTTP verbs, we now also need to use HTTP return codes properly. For example, if the order was not found we return HTTP 404 "Not Found".

This error response does not necessarily need to contain any body message, however for usability purposes it is still preferred to return some error details, such as:

**Example XML**

<Errors>
  <Error>
    <Type>13</Type>
    <Code>APP_ERR</Code>
    <Description>DisplayBooking: 10054 Exception Caught While Retrieving Booking : NO MATCH FOUND</Description>
  </Error>
</Errors>
Similarly other requests that, for example, add a bag to a passenger on a given flight will return HTTP 201 “Created” if the bag was successfully added or HTTP 403 “Forbidden” if the caller is not able to add a bag, because check-in is already closed for example.

### 13.4 RMM Level 3 NDC Example

To support RESTful architecture at this level further modifications will be required to all RS messages. In the same scenario as was shown in the examples for the previous level, we retrieve an order using the same request:

**Example URL**

```
GET /orders/bookings/ORDER-c98c-4759-a20b
```

But if the retrieval was successful, we would get the following message back:

**Example XML**

```
<Order xmlns="http://www.iata.org/IATA/EDIST">
  <Link Rel="self" Url="/orders/bookings/ORDER-c98c-4759-a20b"/>
  <Link Rel="http://iata.org/schemas/order/payment" Url="/orders/bookings/ORDER-c98c-4759-a20b/payments"/>
  <Link Rel="http://iata.org/schemas/order/passengers" Url="/orders/bookings/ORDER-c98c-4759-a20b/passengers"/>
  <Passengers>
    <Passenger ObjectKey="PAX1">
      <Link Rel="self" Url="/orders/bookings/ORDER-c98c-4759-a20b/passengers/PAX1"/>
      <Link Rel="http://iata.org/schemas/order/passenger/bags" Url="/orders/orders/ORDER-c98c-4759-a20b/passengers/PAX1/bags"/>
      <PTC>ADT</PTC>
      <Name></Name>
        <Surname>MUSTERMANN</Surname>
        <Given>MAX</Given>
        <Title>MR</Title>
      </Name>
      <Gender>Male</Gender>
    </Passenger>
  </Passengers>
</Order>
```
Example XML

```xml

</Passenger>
</Passengers>

</Order>
```

Example JSON

```json

{
  "Order": {
    "Link": [{
      "Rel": "self",
      "Uri": "/orders/bookings/ORDER-c98c-4759-a20b"
    }, {
      "Rel": "http://iata.org/schemas/order/payment",
      "Uri": "/orders/bookings/ORDER-c98c-4759-a20b/payments"
    }, {
      "Rel": "http://iata.org/schemas/order/passengers",
      "Uri": "/orders/bookings/ORDER-c98c-4759-a20b/passengers"
    }],
    "Passengers": [{
      "Link": [{
        "Rel": "self",
        "Uri": "/orders/bookings/ORDER-c98c-4759-a20b/passengers/PAX1"
      }, {
        "Rel": "http://iata.org/schemas/order/passenger/bags",
        "Uri": "/orders/bookings/ORDER-c98c-4759-a20b/passengers/PAX1/bags"
      }],
      "Gender": "Male",
      "Name": {
        "Surname": "MUSTERMANN",
        "Given": "MAX",
        "Title": "MR"
      },
      "PTC": "ADT",
      "ObjectKey": "PAX1"
    }]
  }
}
```

Note here the usage of the link tag to provide the details on each level of what can be done with the resource or what this resource represents (Rel attribute), and which URL should be used during this action (Uri attribute).

Now we can clearly see what actions we can execute next and what URIs we have to use. For example, in order to add baggage we would need to POST to `/orders/bookings/ORDER-C98C-4759-A20B/passengers/PAX1/bags`, or if the baggage needs to be removed to execute a DELETE on that URL. The advantage of using this approach is than now as a consumer of the service, we do not need to know the logic of what URI to use for which resource or how to convert any identifier that may be present in the document in order to retrieve the corresponding information. There is also an advantage for the service provider, in that any changes to the structure of the URI can be done transparently to consumers or new actions that may be executed on the resource can be added when required.
Additionally, link with the Rel="self" attribute is a convention to provide a link that points to the resource itself.

### Possible XSD schema modification

Make sure all "xsd:element" definitions that represent a resource accessible via URL contain an optional list of link elements as shown above.

### 13.5 General considerations adopting JSON message format

#### 13.5.1 Use of XML Attributes

XML has the ability to store data either in attributes or as a tag value. This concept does not directly map to JSON format, since in JSON all object properties are always key-value pairs.

The current NDC schemas extensively use both attributes and tag values. If we convert the same message to a JSON format, additional nodes will be added to the JSON message to handle this situation. For example, for the following XML:

**Example XML**

```
<PointOfSale>
  <Location>
    <CountryCode>CH</CountryCode>
    <CityCode>ZRH</CityCode>
  </Location>
</PointOfSale>
```

We would expect the following simple JSON message:

**Example JSON**

```
{
  "PointOfSale":{
    "Location":{
      "CountryCode":"CH",
      "CityCode":"ZRH"
    }
  }
}
```

However, because "CityCode" is defined in an NDC schema as a complex element that apart from an actual value can also contain "ObjectMetaReferences" and "refs" attributes like "<CityCode ref='xxx' ObjectMetaReferences='meta1'>ZRH</CityCode>", we actually get the following JSON representation:

**Example JSON**

```
{
  "PointOfSale":{
    "Location":{
      "CountryCode":"CH",
      "CityCode":{
        "Value":"ZRH"
      }
    }
  }
}
```
Note the extra value attribute that does not exist in the schema but is present in the JSON representation. This is not really a technical problem, but adds an additional level to JSON message structure which makes it overall less usable.

Another issue that could arise when JSON messages are used, is the name conflict, when both attribute name and tag name have the same value like in FlightDepartureType and FlightArrivalType NDC complex types. They both have an element named Time and an attribute with the same name.

This can not be directly converted to JSON and depending on a JSON library used to generate code bindings, it will rename one of the names to a unique name like:

```
"Departure": {
  "AirportCode": {
    "Value": "ZRH"
  },
  "Date": "2015-03-28",
  "Time": "10:20",
  "Time1": "07:05"
}
```

Here we have original departure Time stored as Time member but Time attribute was automatically renamed to a Time1 member.

To avoid JSON name conflicts between XML attribute names and XML element names make sure all NDC elements do not contain same names in both attributes and immediate child elements like in FlightDepartureType and FlightArrivalType

### 13.5.2 Encoding special characters

Unlike XML, JSON messages do not have the ability to support CDATA sections. This means that during serialization one should always escape all special characters in text values to JSON specification as defined on [http://json.org/](http://json.org/)
13.5.3 Validating JSON documents
Unlike XML, where validations against XML schema definitions can be done, there is no well adopted standard for validating JSON messages. One option that currently exists is the use of JSON schemas for validation [4, 27]. While there are several tools and libraries that can validate JSON messages against JSON schema definitions, they seem not to be widely used yet. In addition, there is no standard way to convert XSD documents to JSON schema documents. Some libraries like JSON.NET can, however, generate JSON schema documents from the annotated source code.

13.5.4 Using XML namespaces
JSON does not have a concept of namespaces as they exist in the XML. Current NDC schemas always use the same namespace for all messages, so when serializing it to JSON this namespace can be ignored. However, if IATA decides to introduce different namespaces at a later stage this will require special handling on the JSON side, similar to object type handling using the “$type” attribute described above. This will introduce additional complexity and require custom processing and should therefore be avoided.