Microsoft REST API Guidelines

Microsoft REST API Guidelines Working Group

Name	Name	Name
Dave Campbell (CTO C+E)	Rick Rashid (CTO ASG)	John Shewchuk (Technical Fellow, TED HQ)
Mark Russinovich (CTO Azure)	Steve Lucco (Technical Fellow, DevDiv)	Murali Krishnaprasad (Azure App Plat)
Rob Howard (ASG)	Peter Torr (OSG)	Chris Mullins (ASG)

Document editors: John Gossman (C+E), Chris Mullins (ASG), Gareth Jones (ASG), Rob Dolin (C+E), Mark Stafford (C+E)

Microsoft REST API Guidelines

1. Abstract

The Microsoft REST API Guidelines, as a design principle, encourages application developers to have resources accessible to them via a RESTful HTTP interface. To provide the smoothest possible experience for developers on platforms following the Microsoft REST API Guidelines, REST APIs SHOULD follow consistent design guidelines to make using them easy and intuitive.

This document establishes the guidelines Microsoft REST APIs SHOULD follow so RESTful interfaces are developed consistently.

2. Table of contents

3. Introduction

Developers access most Microsoft Cloud Platform resources via HTTP interfaces. Although each service typically provides language-specific frameworks to wrap their APIs, all of their operations eventually boil down to HTTP requests. Microsoft must support a wide range of clients and services and cannot rely on rich frameworks being available for every development environment. Thus, a goal of these guidelines is to ensure Microsoft REST APIs can be easily and consistently consumed by any client with basic HTTP support.

To provide the smoothest possible experience for developers, it's important to have these APIs follow consistent design guidelines, thus making using them easy and intuitive. This document establishes the guidelines to be followed by Microsoft REST API developers for developing such APIs consistently.

The benefits of consistency accrue in aggregate as well; consistency allows teams to leverage common code, patterns, documentation and design decisions.

These guidelines aim to achieve the following: - Define consistent practices and patterns for all API endpoints across Microsoft. - Adhere as closely as possible to accepted REST/HTTP best practices in the industry at-large. [*] - Make accessing Microsoft Services via REST interfaces easy for all application developers. - Allow service developers to leverage the prior work of other services to implement, test and document REST endpoints defined consistently. - Allow for partners (e.g., non-Microsoft entities) to use these guidelines for their own REST endpoint design.

[*] Note: The guidelines are designed to align with building services which comply with the REST architectural style, though they do not address or require building services that follow the REST constraints. The term "REST" is used throughout this document to mean services that are in the spirit of REST rather than adhering to REST by the book.*

3.1. Recommended reading

Understanding the philosophy behind the REST Architectural Style is recommended for developing good HTTP-based services. If you are new to RESTful design, here are some good resources:

REST on Wikipedia -- Overview of common definitions and core ideas behind REST.

REST Dissertation -- The chapter on REST in Roy Fielding's dissertation on Network Architecture, "Architectural Styles and the Design of Network-based Software Architectures"

RFC 7231 -- Defines the specification for HTTP/1.1 semantics, and is considered the authoritative resource.

REST in Practice -- Book on the fundamentals of REST.

4. Interpreting the guidelines

4.1. Application of the guidelines

These guidelines are applicable to any REST API exposed publicly by Microsoft or any partner service. Private or internal APIs SHOULD also try to follow these guidelines because internal services tend to eventually be exposed publicly. Consistency is valuable to not only external customers but also internal service consumers, and these guidelines offer best practices useful for any service.

There are legitimate reasons for exemption from these guidelines. Obviously, a REST service that implements or must interoperate with some externally defined REST API must be compatible with that API and not necessarily these guidelines. Some services MAY also have special performance needs that require a different format, such as a binary protocol.

4.2. Guidelines for existing services and versioning of services

We do not recommend making a breaking change to a service that predates these guidelines simply for the sake of compliance. The service SHOULD try to become compliant at the next version release when compatibility is being broken anyway. When a service adds a new API, that API SHOULD be consistent with the other APIs of the same version. So if a service was written against version 1.0 of the guidelines, new APIs added incrementally to the service SHOULD also follow version 1.0. The service can then upgrade to align with the latest version of the guidelines at the service's next major release.

4.3. Requirements language

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

4.4. License

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

5. Taxonomy

As part of onboarding to Microsoft REST API Guidelines, services MUST comply with the taxonomy defined below.

5.1. Errors

Errors, or more specifically Service Errors, are defined as a client passing invalid data to the service and the service *correctly* rejecting that data. Examples include invalid credentials, incorrect parameters, unknown version IDs, or similar. These are generally "4xx" HTTP error codes and are the result of a client passing incorrect or invalid data.

Errors do *not* contribute to overall API availability.

5.2. Faults

Faults, or more specifically Service Faults, are defined as the service failing to correctly return in response to a valid client request. These are generally "5xx" HTTP error codes.

Faults *do* contribute to the overall API availability.

Calls that fail due to rate limiting or quota failures MUST NOT count as faults. Calls that fail as the result of a service fast-failing requests (often for its own protection) do count as faults.

5.3. Latency

Latency is defined as how long a particular API call takes to complete, measured as closely to the client as possible. This metric applies to both synchronous and asynchronous APIs in the same way. For long running calls, the latency is measured on the initial request and measures how long that call (not the overall operation) takes to complete.

5.4. Time to complete

Services that expose long operations MUST track "Time to Complete" metrics around those operations.

5.5. Long running API faults

For a Long Running API, it's possible for both the initial request which begins the operation and the request which retrieves the results to technically work (each passing back a 200) but for the underlying operation to have failed. Long Running faults MUST roll up as faults into the overall Availability metrics.

6. Client guidance

To ensure the best possible experience for clients talking to a REST service, clients SHOULD adhere to the following best practices:

6.1. Ignore rule

For loosely coupled clients where the exact shape of the data is not known before the call, if the server returns something the client wasn't expecting, the client MUST safely ignore it.

Some services MAY add fields to responses without changing versions numbers. Services that do so MUST make this clear in their documentation and clients MUST ignore unknown fields.

6.2. Variable order rule

Clients MUST NOT rely on the order in which data appears in JSON service responses. For example, clients SHOULD be resilient to the reordering of fields within a JSON object. When supported by the service, clients MAY request that data be returned in a specific order. For example, services MAY support the use of the *$orderBy* querystring parameter to specify the order of elements within a JSON array. Services MAY also explicitly specify the ordering of some elements as part of the service contract. For example, a service MAY always return a JSON object's "type" information as the first field in an object to simplify response parsing on the client. Clients MAY rely on ordering behavior explicitly identified by the service.

6.3. Silent fail rule

Clients requesting OPTIONAL server functionality (such as optional headers) MUST be resilient to the server ignoring that particular functionality.

7. Consistency fundamentals

7.1. URL structure

Humans SHOULD be able to easily read and construct URLs.

This facilitates discovery and eases adoption on platforms without a well-supported client library.

An example of a well-structured URL is:

```
https://api.contoso.com/v1.0/people/jdoe@contoso.com/inbox
```

An example URL that is not friendly is:

```
https://api.contoso.com/EWS/OData/Users('jdoe@microsoft.com')/Folders('AAMkADdiYzI1MjUz
```

A frequent pattern that comes up is the use of URLs as values. Services MAY use URLs as values. For example, the following is acceptable:

```
https://api.contoso.com/v1.0/items?url=https://resources.contoso.com/shoes/fancy
```

7.2. URL length

The HTTP 1.1 message format, defined in RFC 7230, in section 3.1.1, defines no length limit on the Request Line, which includes the target URL. From the RFC:

> HTTP does not place a predefined limit on the length of a request-line. [...] A server that receives a request-target longer than any URI it wishes to parse MUST respond with a 414 (URI Too Long) status code.

Services that can generate URLs longer than 2,083 characters MUST make accommodations for the clients they wish to support. Here are some sources for determining what target clients support:

- http://stackoverflow.com/a/417184
- https://blogs.msdn.microsoft.com/ieinternals/2014/08/13/url-length-limits/

Also note that some technology stacks have hard and adjustable URL limits, so keep this in mind as you design your services.

7.3. Canonical identifier

In addition to friendly URLs, resources that can be moved or be renamed SHOULD expose a URL that contains a unique stable identifier. It MAY be necessary to interact with the service to obtain a stable URL from the friendly name for the resource, as in the case of the "/my" shortcut used by some services.

The stable identifier is not required to be a GUID.

An example of a URL containing a canonical identifier is:

```
https://api.contoso.com/v1.0/people/7011042402/inbox
```

7.4. Supported methods

Operations MUST use the proper HTTP methods whenever possible, and operation idempotency MUST be respected. HTTP methods are frequently referred to as the HTTP verbs. The terms are synonymous in this context, however the HTTP specification uses the term method.

Below is a list of methods that Microsoft REST services SHOULD support. Not all resources will support all methods, but all resources using the methods below MUST conform to their usage.

Method	Description	Is Idempotent
GET	Return the current value of an object	True
PUT	Replace an object, or create a named object, when applicable	True
DELETE	Delete an object	True
POST	Create a new object based on the data provided, or submit a command	False
HEAD	Return metadata of an object for a GET response. Resources that support the GET method MAY support the HEAD method as well	True
PATCH	Apply a partial update to an object	False
OPTIONS	Get information about a request; see below for details.	True

Table 1

7.4.1. POST

POST operations SHOULD support the Location response header to specify the location of any created resource that was not explicitly named, via the Location header.

As an example, imagine a service that allows creation of hosted servers, which will be named by the service:

```
POST http://api.contoso.com/account1/servers
```

The response would be something like:

```
 201 Created
Location: http://api.contoso.com/account1/servers/server321
```

Where "server321" is the service-allocated server name.

Services MAY also return the full metadata for the created item in the response.

7.4.2. PATCH

PATCH has been standardized by IETF as the method to be used for updating an existing object incrementally (see RFC 5789). Microsoft REST API Guidelines compliant APIs SHOULD support PATCH.

7.4.3. Creating resources via PATCH (UPSERT semantics)

Services that allow callers to specify key values on create SHOULD support UPSERT semantics, and those that do MUST support creating resources using PATCH. Because PUT is defined as a complete replacement of the content, it is dangerous for clients to use PUT to modify data. Clients that do not understand (and hence ignore) properties on a resource are not likely to provide them on a PUT when trying to update a resource, hence such properties could be inadvertently removed. Services MAY optionally support PUT to update existing resources, but if they do they MUST use replacement semantics (that is, after the PUT, the resource's properties MUST match what was provided in the request, including deleting any server properties that were not provided).

Under UPSERT semantics, a PATCH call to a nonexistent resource is handled by the server as a "create," and a PATCH call to an existing resource is handled as an "update." To ensure that an update request is not treated as a create or vice versa, the client MAY specify precondition HTTP headers in the request. The service MUST NOT treat a PATCH request as an insert if it contains an If-Match header and MUST NOT treat a PATCH request as an update if it contains an If-None-Match header with a value of "*".

If a service does not support UPSERT, then a PATCH call against a resource that does not exist MUST result in an HTTP "409 Conflict" error.

7.4.4. Options and link headers

OPTIONS allows a client to retrieve information about a resource, at a minimum by returning the Allow header denoting the valid methods for this resource.

In addition, services SHOULD include a Link header (seeRFC 5988) to point to documentation for the resource in question:

```
Link: <{help}>; rel="help"
```

Where {help} is the URL to a documentation resource.

For examples on use of OPTIONS, seepreflighting CORS cross-domain calls.

7.5. Standard request headers

The table of request headers below SHOULD be used by Microsoft REST API Guidelines services. Using these headers is not mandated, but if used they MUST be used consistently.

All header values MUST follow the syntax rules set forth in the specification where the header field is defined. Many HTTP headers are defined in RFC7231, however a complete list of approved headers can be found in the IANA Header Registry."

Header	Type	Description
Authorization	String	Authorization header for the request

Header	Type	Description
Date	Date	Timestamp of the request, based on the client's clock, in RFC 5322 date and time format. The server SHOULD NOT make any assumptions about the accuracy of the client's clock. This header MAY be included in the request, but MUST be in this format when supplied. Greenwich Mean Time (GMT) MUST be used as the time zone reference for this header when it is provided. For example: `Wed, 24 Aug 2016 18:41:30 GMT`. Note that GMT is exactly equal to UTC (Coordinated Universal Time) for this purpose.
Accept	Content type	The requested content type for the response such as:

- application/xml
- text/xml
- application/json
- text/javascript (for JSONP)

Per the HTTP guidelines, this is just a hint and responses MAY have a different content type, such as a blob fetch where a successful response will just be the blob stream as the payload. For services following OData, the preference order specified in OData SHOULD be followed. Accept-Encoding | Gzip, deflate | REST endpoints SHOULD support GZIP and DEFLATE encoding, when applicable. For very large resources, services MAY ignore and return uncompressed data. Accept-Language | "en", "es", etc. | Specifies the preferred language for the response. Services are not required to support this, but if a service supports localization it MUST do so through the Accept-Language header. Accept-Charset | Charset type like "UTF-8" | Default is UTF-8, but services SHOULD be able to handle ISO-8859-1. Content-Type | Content type | Mime type of request body (PUT/POST/PATCH) Prefer | return=minimal, return=representation | If the return=minimal preference is specified, services SHOULD return an empty body in response to a successful insert or update. If return=representation is specified, services SHOULD return the created or updated resource in the response. Services SHOULD support this header if they have scenarios where clients would sometimes benefit from responses, but sometimes the response would impose too much of a hit on bandwidth. If-Match, If-None-Match, If-Range | String | Services that support updates to resources using optimistic concurrency control MUST support the If-Match header to do so. Services MAY also use other headers related to ETags as long as they follow the HTTP specification.

7.6. Standard response headers

Services SHOULD return the following response headers, except where noted in the "required" column.

Response Header	Required	Description
Date	All responses	Timestamp the response was processed, based on the server's clock, in RFC 5322 date and time format. This header MUST be included in the response. Greenwich Mean Time (GMT) MUST be used as the time zone reference for this header. For example: `Wed, 24 Aug 2016 18:41:30 GMT`. Note that GMT is exactly equal to UTC (Coordinated Universal Time) for this purpose.
Content-Type	All responses	The content type

Response Header	Required	Description
Content-Encoding	All responses	GZIP or DEFLATE, as appropriate
Preference-Applied	When specified in request	Whether a preference indicated in the Prefer request header was applied
ETag	When the requested resource has an entity tag	The ETag response-header field provides the current value of the entity tag for the requested variant. Used with If-Match, If-None-Match and If-Range to implement optimistic concurrency control.

7.7. Custom headers

Custom headers MUST NOT be required for the basic operation of a given API.

Some of the guidelines in this document prescribe the use of nonstandard HTTP headers. In addition, some services MAY need to add extra functionality, which is exposed via HTTP headers. The following guidelines help maintain consistency across usage of custom headers.

Headers that are not standard HTTP headers MUST have one of two formats:

1. A generic format for headers that are registered as "provisional" with IANA (RFC 3864)
2. A scoped format for headers that are too usage-specific for registration

These two formats are described below.

7.8. Specifying headers as query parameters

Some headers pose challenges for some scenarios such as AJAX clients, especially when making cross-domain calls where adding headers MAY not be supported. As such, some headers MAY be accepted as Query Parameters in addition to headers, with the same naming as the header:

Not all headers make sense as query parameters, including most standard HTTP headers.

The criteria for considering when to accept headers as parameters are:

1. Any custom headers MUST be also accepted as parameters.
2. Required standard headers MAY be accepted as parameters.
3. Required headers with security sensitivity (e.g., Authorization header) MIGHT NOT be appropriate as parameters; the service owner SHOULD evaluate these on a case-by-case basis.

The one exception to this rule is the Accept header. It's common practice to use a scheme with simple names instead of the full functionality described in the HTTP specification for Accept.

7.9. PII parameters

Consistent with their organization's privacy policy, clients SHOULD NOT transmit personally identifiable information (PII) parameters in the URL (as part of path or query string) because this information can be inadvertently exposed via client, network, and server logs and other mechanisms.

Consequently, a service SHOULD accept PII parameters transmitted as headers.

However, there are many scenarios where the above recommendations cannot be followed due to client or software limitations. To address these limitations, services SHOULD also accept these PII parameters as part of the URL consistent with the rest of these guidelines.

Services that accept PII parameters -- whether in the URL or as headers -- SHOULD be compliant with privacy policy specified by their organization's engineering leadership. This will typically include recommending that clients prefer headers for transmission and implementations adhere to special precautions to ensure that logs and other service data collection are properly handled.

7.10. Response formats

For organizations to have a successful platform, they must serve data in formats developers are accustomed to using, and in consistent ways that allow developers to handle responses with common code.

Web-based communication, especially when a mobile or other low-bandwidth client is involved, has moved quickly in the direction of JSON for a variety of reasons, including its tendency to be lighter weight and its ease of consumption with JavaScript-based clients.

JSON property names SHOULD be camelCased.

Services SHOULD provide JSON as the default encoding.

7.10.1. Clients-specified response format

In HTTP, response format SHOULD be requested by the client using the Accept header. This is a hint, and the server MAY ignore it if it chooses to, even if this isn't typical of well-behaved servers. Clients MAY send multiple Accept headers and the service MAY choose one of them.

The default response format (no Accept header provided) SHOULD be application/json, and all services MUST support application/json.

Accept Header	Response type	Notes
application/json	Payload SHOULD be returned as JSON	Also accept text/javascript for JSONP cases

```
 GET https://api.contoso.com/v1.0/products/user
Accept: application/json
```

7.10.2. Error condition responses

For non-success conditions, developers SHOULD be able to write one piece of code that handles errors consistently across different Microsoft REST API Guidelines services. This allows building of simple and reliable infrastructure to handle exceptions as a separate flow from successful responses. The following is based on the OData v4 JSON spec. However, it is very generic and does not require specific OData constructs. APIs SHOULD use this format even if they are not using other OData constructs.

The error response MUST be a single JSON object. This object MUST have a name/value pair named "error." The value MUST be a JSON object.

This object MUST contain name/value pairs with the names "code" and "message," and it MAY contain name/value pairs with the names "target," "details" and "innererror."

The value for the "code" name/value pair is a language-independent string. Its value is a service-defined error code that SHOULD be human-readable. This code serves as a more specific indicator of the error than the HTTP error code specified in the response. Services SHOULD have a relatively small number (about 20) of possible values for "code," and all clients MUST be capable of handling all of them. Most services will require a much larger number of more specific error codes, which are not interesting to all clients. These error codes SHOULD be exposed in the "innererror" name/value pair as described below. Introducing a new value for "code" that is visible to existing clients is a breaking change and requires a version increase. Services can avoid breaking changes by adding new error codes to "innererror" instead.

The value for the "message" name/value pair MUST be a human-readable representation of the error. It is intended as an aid to developers and is not suitable for exposure to end users. Services wanting to expose a suitable message for end users MUST do so through an annotation or custom property. Services SHOULD NOT localize "message" for the end user, because doing so might make the value unreadable to the app developer who may be logging the value, as well as make the value less searchable on the Internet.

The value for the "target" name/value pair is the target of the particular error (e.g., the name of the property in error).

The value for the "details" name/value pair MUST be an array of JSON objects that MUST contain name/value pairs for "code" and "message," and MAY contain a name/value pair for "target," as described above. The objects in the "details" array usually represent distinct, related errors that occurred during the request. See example below.

The value for the "innererror" name/value pair MUST be an object. The contents of this object are service-defined. Services wanting to return more specific errors than the root-level code MUST do so by including a name/value pair for "code" and a nested "innererror." Each nested "innererror" object represents a higher level of detail than its parent. When evaluating errors, clients MUST traverse through all of the nested "innererrors" and choose the deepest one that they understand. This scheme allows services to introduce new error codes anywhere in the hierarchy without breaking backwards compatibility, so long as old error codes still appear. The service MAY return different levels of depth and detail to different callers. For example, in development environments, the deepest "innererror" MAY contain internal information that can help debug the service. To guard against potential security concerns around information disclosure, services SHOULD take care not to expose too much detail unintentionally. Error objects MAY also include custom server-defined name/value pairs that MAY be specific to the code. Error types with custom server-defined properties SHOULD be declared in the service's metadata document. See example below.

Error responses MAY contain annotations in any of their JSON objects.

We recommend that for any transient errors that may be retried, services SHOULD include a Retry-After HTTP header indicating the minimum number of seconds that clients SHOULD wait before attempting the operation again.

ErrorResponse : Object

Property	Type	Required	Description
`error`	Error	✔	The error object.

Error : Object

Property	Type	Required	Description
`code`	String	✔	One of a server-defined set of error codes.
`message`	String	✔	A human-readable representation of the error.
`target`	String		The target of the error.
`details`	Error[]		An array of details about specific errors that led to this reported error.
`innererror`	InnerError		An object containing more specific information than the current object about the error.

InnerError : Object

Property	Type	Required	Description
`code`	String		A more specific error code than was provided by the containing error.
`innererror`	InnerError		An object containing more specific information than the current object about the error.

Examples

Example of "innererror":

```
{
  "error": {
    "code": "BadArgument",
    "message": "Previous passwords may not be reused",
    "target": "password",
    "innererror": {
      "code": "PasswordError",
      "innererror": {
        "code": "PasswordDoesNotMeetPolicy",
        "minLength": "6",
        "maxLength": "64",
        "characterTypes": ["lowerCase","upperCase","number","symbol"],
        "minDistinctCharacterTypes": "2",
        "innererror": {
          "code": "PasswordReuseNotAllowed"
        }
      }
    }
  }
}
```

In this example, the most basic error code is "BadArgument," but for clients that are interested, there are more specific error codes in "innererror." The "PasswordReuseNotAllowed" code may have been added by the service at a later date, having previously only returned "PasswordDoesNotMeetPolicy." Existing clients do not break when the new error code is added, but new clients MAY take advantage of it. The "PasswordDoesNotMeetPolicy" error also includes additional name/value pairs that allow the client to determine the server's configuration, validate the user's input programmatically, or present the server's constraints to the user within the client's own localized messaging.

Example of "details":

```
{
  "error": {
    "code": "BadArgument",
    "message": "Multiple errors in ContactInfo data",
    "target": "ContactInfo",
    "details": [
      {
        "code": "NullValue",
        "target": "PhoneNumber",
        "message": "Phone number must not be null"
      },
      {
        "code": "NullValue",
        "target": "LastName",
        "message": "Last name must not be null"
      },
      {
        "code": "MalformedValue",
        "target": "Address",
        "message": "Address is not valid"
      }
    ]
  }
}
```

In this example there were multiple problems with the request, with each individual error listed in "details."

7.11. HTTP Status Codes

Standard HTTP Status Codes SHOULD be used; see the HTTP Status Code definitions for more information.

7.12. Client library optional

Developers MUST be able to develop on a wide variety of platforms and languages, such as Windows, macOS, Linux, C#, Python, Node.js, and Ruby.

Services SHOULD be able to be accessed from simple HTTP tools such as curl without significant effort.

Service developer portals SHOULD provide the equivalent of "Get Developer Token" to facilitate experimentation and curl support.

8. CORS

Services compliant with the Microsoft REST API Guidelines MUST supportCORS (Cross Origin Resource Sharing). Services SHOULD support an allowed origin of CORS * and enforce authorization through valid OAuth tokens. Services SHOULD NOT support user credentials with origin validation. There MAY be exceptions for special cases.

8.1. Client guidance

Web developers usually don't need to do anything special to take advantage of CORS. All of the handshake steps happen invisibly as part of the standard XMLHttpRequest calls they make.

Many other platforms, such as .NET, have integrated support for CORS.

8.1.1. Avoiding preflight

Because the CORS protocol can trigger preflight requests that add additional round trips to the server, performance-critical apps might be interested in avoiding them. The spirit behind CORS is to avoid preflight for any simple cross-domain requests that old non-CORS-capable browsers were able to make. All other requests require preflight.

A request is "simple" and avoids preflight if its method is GET, HEAD or POST, and if it doesn't contain any request headers besides Accept, Accept-Language and Content-Language. For POST requests, the Content-Type header is also allowed, but only if its value is "application/x-www-form-urlencoded," "multipart/form-data" or "text/plain." For any other headers or values, a preflight request will happen.

8.2. Service guidance

At minimum, services MUST: - Understand the Origin request header that browsers send on cross-domain requests, and the Access-Control-Request-Method request header that they send on preflight OPTIONS requests that check for access. - If the Origin header is present in a request: - If the request uses the OPTIONS method and contains the Access-Control-Request-Method header, then it is a preflight request intended to probe for access before the actual request. Otherwise, it is an actual request. For preflight requests, beyond performing the steps below to add headers, services MUST perform no additional processing and MUST return a 200 OK. For non-preflight requests, the headers below are added in addition to the request's regular processing. - Add an Access-Control-Allow-Origin header to the response, containing the same value as the Origin request header. Note that this requires services to dynamically generate the header value. Resources that do not require cookies or any other form of user credentials MAY respond with a wildcard asterisk (*) instead. Note that the wildcard is acceptable here only, and not for any of the other headers described below. - If the caller requires access to a response header that is not in the set of simple response headers (Cache-Control, Content-Language, Content-Type, Expires, Last-Modified, Pragma), then add an Access-Control-Expose-Headers header containing the list of additional response header names the client should have access to. - If the request requires cookies, then add an Access-Control-Allow-Credentials header set to "true." - If the request was a preflight request (see first bullet), then the service MUST: - Add an Access-Control-Allow-Headers response header containing the list of request header names the client is permitted to use. This list need only contain headers that are not in the set of simple request headers (Accept, Accept-Language, Content-Language). If there are no restrictions on headers the service accepts, the service MAY simply return the same value as the Access-Control-Request-Headers header sent by the client. - Add an Access-Control-Allow-Methods response header containing the list of HTTP methods the caller is permitted to use.

Add an Access-Control-Max-Age pref response header containing the number of seconds for which this preflight response is valid (and hence can be avoided before subsequent actual requests). Note that while it is customary to use a large value like 2592000 (30 days), many browsers self-impose a much lower limit (e.g., five minutes).

Because browser preflight response caches are notoriously weak, the additional round trip from a preflight response hurts performance. Services used by interactive Web clients where performance is critical SHOULD avoid patterns that cause a preflight request - For GET and HEAD calls, avoid requiring request headers that are not part of the simple set above. Allow them to be provided as query parameters instead. - The Authorization header is not part of the simple set, so the authentication token MUST be sent through the "access_token" query parameter instead, for resources requiring authentication. Note that passing authentication tokens in the URL is not recommended, because it can lead to the token getting recorded in server logs and exposed to anyone with access to those logs. Services that accept authentication tokens through the URL MUST take steps to mitigate the security risks, such as using short-lived authentication tokens, suppressing the auth token from getting logged, and controlling access to server logs.

- Avoid requiring cookies. XmlHttpRequest will only send cookies on cross-domain requests if the "withCredentials" attribute is set; this also causes a preflight request.
    - Services that require cookie-based authentication MUST use a "dynamic canary" to secure all APIs that accept cookies.
- For POST calls, prefer simple Content-Types in the set of ("application/x-www-form-urlencoded," "multipart/form-data," "text/plain") where applicable. Any other Content-Type will induce a preflight request.
    - Services MUST NOT contravene other API recommendations in the name of avoiding CORS preflight requests. In particular, in accordance with recommendations, most POST requests will actually require a preflight request due to the Content-Type.
    - If eliminating preflight is critical, then a service MAY support alternative mechanisms for data transfer, but the RECOMMENDED approach MUST also be supported.

In addition, when appropriate services MAY support the JSONP pattern for simple, GET-only cross-domain access. In JSONP, services take a parameter indicating the format (*$format=json*) and a parameter indicating a callback (*$callback=someFunc*), and return a text/javascript document containing the JSON response wrapped in a function call with the indicated name. More on JSONP at Wikipedia: JSONP.

9. Collections

9.1. Item keys

Services MAY support durable identifiers for each item in the collection, and that identifier SHOULD be represented in JSON as "id". These durable identifiers are often used as item keys.

Collections that support durable identifiers MAY support delta queries.

9.2. Serialization

Collections are represented in JSON using standard array notation.

9.3. Collection URL patterns

Collections are located directly under the service root when they are top level, or as a segment under another resource when scoped to that resource.

For example:

```
GET https://api.contoso.com/v1.0/people
```

Whenever possible, services MUST support the "/" pattern. For example:

```
GET https://{serviceRoot}/{collection}/{id}
```

Where: - {serviceRoot} – the combination of host (site URL) + the root path to the service - {collection} – the name of the collection, unabbreviated, pluralized - {id} – the value of the unique id property. When using the "/" pattern this MUST be the raw string/number/guid value with no quoting but properly escaped to fit in a URL segment.

9.3.1. Nested collections and properties

Collection items MAY contain other collections. For example, a user collection MAY contain user resources that have multiple addresses:

```
GET https://api.contoso.com/v1.0/people/123/addresses
```

```
{
  "value": [
    { "street": "1st Avenue", "city": "Seattle" },
    { "street": "124th Ave NE", "city": "Redmond" }
  ]
}
```

9.4. Big collections

As data grows, so do collections. Planning for pagination is important for all services. Therefore, when multiple pages are available, the serialization payload MUST contain the opaque URL for the next page as appropriate. Refer to the paging guidance for more details.

Clients MUST be resilient to collection data being either paged or nonpaged for any given request.

```
{
  "value":[
    { "id": "Item 1","price": 99.95,"sizes": null},
    { … },
    { … },
    { "id": "Item 99","price": 59.99,"sizes": null}
  ],
  "@nextLink": "{opaqueUrl}"
}
```

9.5. Changing collections

POST requests are not idempotent. This means that two POST requests sent to a collection resource with exactly the same payload MAY lead to multiple items being created in that collection. This is often the case for insert operations on items with a server-side generated id.

For example, the following request:

```
POST https://api.contoso.com/v1.0/people
```

Would lead to a response indicating the location of the new collection item:

```
201 Created
Location: https://api.contoso.com/v1.0/people/123
```

And once executed again, would likely lead to another resource:

```
201 Created
Location: https://api.contoso.com/v1.0/people/124
```

While a PUT request would require the indication of the collection item with the corresponding key instead:

```
PUT https://api.contoso.com/v1.0/people/123
```

9.6. Sorting collections

The results of a collection query MAY be sorted based on property values. The property is determined by the value of the *$orderBy* query parameter.

The value of the *$orderBy* parameter contains a comma-separated list of expressions used to sort the items. A special case of such an expression is a property path terminating on a primitive property.

The expression MAY include the suffix "asc" for ascending or "desc" for descending, separated from the property name by one or more spaces. If "asc" or "desc" is not specified, the service MUST order by the specified property in ascending order.

NULL values MUST sort as "less than" non-NULL values.

Items MUST be sorted by the result values of the first expression, and then items with the same value for the first expression are sorted by the result value of the second expression, and so on. The sort order is the inherent order for the type of the property.

For example:

```
GET https://api.contoso.com/v1.0/people?$orderBy=name
```

Will return all people sorted by name in ascending order.

For example:

```
GET https://api.contoso.com/v1.0/people?$orderBy=name desc
```

Will return all people sorted by name in descending order.

Sub-sorts can be specified by a comma-separated list of property names with OPTIONAL direction qualifier.

For example:

```
GET https://api.contoso.com/v1.0/people?$orderBy=name desc,hireDate
```

Will return all people sorted by name in descending order and a secondary sort order of hireDate in ascending order.

Sorting MUST compose with filtering such that:

```
GET https://api.contoso.com/v1.0/people?$filter=name eq 'david'&$orderBy=hireDate
```

Will return all people whose name is David sorted in ascending order by hireDate.

9.6.1. Interpreting a sorting expression

Sorting parameters MUST be consistent across pages, as both client and server-side paging is fully compatible with sorting.

If a service does not support sorting by a property named in a*$orderBy* expression, the service MUST respond with an error message as defined in the Responding to Unsupported Requests section.

9.7. Filtering

The *$filter* querystring parameter allows clients to filter a collection of resources that are addressed by a request URL. The expression specified with *$filter* is evaluated for each resource in the collection, and only items where the expression evaluates to true are included in the response. Resources for which the expression evaluates to false or to null, or which reference properties that are unavailable due to permissions, are omitted from the response.

Example: return all Products whose Price is less than $10.00

```
GET https://api.contoso.com/v1.0/products?$filter=price lt 10.00
```

The value of the *$filter* option is a Boolean expression.

9.7.1. Filter operations

Services that support *$filter* SHOULD support the following minimal set of operations.

Operator	Description	Example
Comparison Operators		
eq	Equal	city eq 'Redmond'
ne	Not equal	city ne 'London'
gt	Greater than	price gt 20
ge	Greater than or equal	price ge 10
lt	Less than	price lt 20
le	Less than or equal	price le 100
Logical Operators		
and	Logical and	price le 200 and price gt 3.5
or	Logical or	price le 3.5 or price gt 200
not	Logical negation	not price le 3.5
Grouping Operators		
( )	Precedence grouping	(priority eq 1 or city eq 'Redmond') and price gt 100

9.7.2. Operator examples

The following examples illustrate the use and semantics of each of the logical operators.

Example: all products with a name equal to 'Milk'

```
GET https://api.contoso.com/v1.0/products?$filter=name eq 'Milk'
```

Example: all products with a name not equal to 'Milk'

```
GET https://api.contoso.com/v1.0/products?$filter=name ne 'Milk'
```

Example: all products with the name 'Milk' that also have a price less than 2.55:

```
GET https://api.contoso.com/v1.0/products?$filter=name eq 'Milk' and price lt 2.55
```

Example: all products that either have the name 'Milk' or have a price less than 2.55:

```
GET https://api.contoso.com/v1.0/products?$filter=name eq 'Milk' or price lt 2.55
```

Example: all products that have the name 'Milk' or 'Eggs' and have a price less than 2.55:

```
GET https://api.contoso.com/v1.0/products?$filter=(name eq 'Milk' or name eq 'Eggs') an
```

9.7.3. Operator precedence

Services MUST use the following operator precedence for supported operators when evaluating*$filter* expressions. Operators are listed by category in order of precedence from highest to lowest. Operators in the same category have equal precedence:

Group	Operator	Description
Grouping	( )	Precedence grouping
Unary	not	Logical Negation
Relational	gt	Greater Than
	ge	Greater than or Equal
	lt	Less Than
	le	Less than or Equal
Equality	eq	Equal
	ne	Not Equal
Conditional AND	and	Logical And
Conditional OR	or	Logical Or

9.8. Pagination

RESTful APIs that return collections MAY return partial sets. Consumers of these services MUST expect partial result sets and correctly page through to retrieve an entire set.

There are two forms of pagination that MAY be supported by RESTful APIs. Server-driven paging mitigates against denial-of-service attacks by forcibly paginating a request over multiple response payloads. Client-driven paging enables clients to request only the number of resources that it can use at a given time.

Sorting and Filtering parameters MUST be consistent across pages, because both client- and server-side paging is fully compatible with both filtering and sorting.

9.8.1. Server-driven paging

Paginated responses MUST indicate a partial result by including a continuation token in the response. The absence of a continuation token means that no additional pages are available.

Clients MUST treat the continuation URL as opaque, which means that query options may not be changed while iterating over a set of partial results.

Example:

```
 GET http://api.contoso.com/v1.0/people HTTP/1.1
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

{
  ...,
  "value": [...],
  "@nextLink": "{opaqueUrl}"
}
```

9.8.2. Client-driven paging

Clients MAY use *$top* and *$skip* query parameters to specify a number of results to return and an offset into the collection.

The server SHOULD honor the values specified by the client; however, clients MUST be prepared to handle responses that contain a different page size or contain a continuation token.

When both *$top* and *$skip* are given by a client, the server SHOULD first apply*$skip* and then *$top* on the collection.

Note: If the server can't honor*$top* and/or *$skip*, the server MUST return an error to the client informing about it instead of just ignoring the query options. This will avoid the risk of the client making assumptions about the data returned.

Example:

```
GET http://api.contoso.com/v1.0/people?$top=5&$skip=2 HTTP/1.1
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

{
  ...,
  "value": [...]
}
```

9.8.3. Additional considerations

Stable order prerequisite: Both forms of paging depend on the collection of items having a stable order. The server MUST supplement any specified order criteria with additional sorts (typically by key) to ensure that items are always ordered consistently.

Missing/repeated results: Even if the server enforces a consistent sort order, results MAY be missing or repeated based on creation or deletion of other resources. Clients MUST be prepared to deal with these discrepancies. The server SHOULD always encode the record ID of the last read record, helping the client in the process of managing repeated/missing results.

Combining client- and server-driven paging: Note that client-driven paging does not preclude server-driven paging. If the page size requested by the client is larger than the default page size supported by the server, the expected response would be the number of results specified by the client, paginated as specified by the server paging settings.

Page Size: Clients MAY request server-driven paging with a specific page size by specifying a*$maxpagesize* preference. The server SHOULD honor this preference if the specified page size is smaller than the server's default page size.

Paginating embedded collections: It is possible for both client-driven paging and server-driven paging to be applied to embedded collections. If a server paginates an embedded collection, it MUST include additional continuation tokens as appropriate.

Recordset count: Developers who want to know the full number of records across all pages, MAY include the query parameter *$count=true* to tell the server to include the count of items in the response.

9.9. Compound collection operations

Filtering, Sorting and Pagination operations MAY all be performed against a given collection. When these operations are performed together, the evaluation order MUST be:

1. **Filtering**. This includes all range expressions performed as an AND operation.
2. **Sorting**. The potentially filtered list is sorted according to the sort criteria.
3. **Pagination**. The materialized paginated view is presented over the filtered, sorted list. This applies to both server-driven pagination and client-driven pagination.

10. Delta queries

Services MAY choose to support delta queries.

10.1. Delta links

Delta links are opaque, service-generated links that the client uses to retrieve subsequent changes to a result.

At a conceptual level delta links are based on a defining query that describes the set of results for which changes are being tracked. The delta link encodes the collection of entities for which changes are being tracked, along with a starting point from which to track changes.

If the query contains a filter, the response MUST include only changes to entities matching the specified criteria. The key principles of the Delta Query are: - Every item in the set MUST have a persistent identifier. That identifier SHOULD be represented as "id". This identifier is a service defined opaque string that MAY be used by the client to track object across calls. - The delta MUST contain an entry for each entity that newly matches the specified criteria, and MUST contain a "@removed" entry for each entity that no longer matches the criteria. - Re-evaluate the query and compare it to original set of results; every entry uniquely in the current set MUST be returned as an Add operation, and every entry uniquely in the original set MUST be returned as a "remove" operation. - Each entity that previously did not match the criteria but matches it now MUST be returned as an "add"; conversely, each entity that previously matched the query but no longer does MUST be returned as a "@removed" entry. - Entities that have changed MUST be included in the set using their standard representation. - Services MAY add additional metadata to the "@removed" node, such as a reason for removal, or a "removed at" timestamp. We recommend teams coordinate with the Microsoft REST API Guidelines Working Group on extensions to help maintain consistency.

The delta link MUST NOT encode any client top or skip value.

10.2. Entity representation

Added and updated entities are represented in the entity set using their standard representation. From the perspective of the set, there is no difference between an added or updated entity.

Removed entities are represented using only their "id" and an "@removed" node. The presence of an "@removed" node MUST represent the removal of the entry from the set.

10.3. Obtaining a delta link

A delta link is obtained by querying a collection or entity and appending a $delta query string parameter. For example:

```
GET https://api.contoso.com/v1.0/people?$delta
HTTP/1.1
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

{
  "value":[
    { "id": "1", "name": "Matt"},
    { "id": "2", "name": "Mark"},
    { "id": "3", "name": "John"}
  ],
  "@deltaLink": "{opaqueUrl}"
}
```

Note: If the collection is paginated the deltaLink will only be present on the final page but MUST reflect any changes to the data returned across all pages.

10.4. Contents of a delta link response

Added/Updated entries MUST appear as regular JSON objects, with regular item properties. Returning the added/modified items in their regular representation allows the client to merge them into their existing "cache" using standard merge concepts based on the "id" field.

Entries removed from the defined collection MUST be included in the response. Items removed from the set MUST be represented using only their "id" and an "@removed" node.

10.5. Using a delta link

The client requests changes by invoking the GET method on the delta link. The client MUST use the delta URL as is -- in other words the client MUST NOT modify the URL in any way (e.g., parsing it and adding additional query string parameters). In this example:

```
 GET https://{opaqueUrl} HTTP/1.1
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

{
  "value":[
    { "id": "1", "name": "Mat"},
    { "id": "2", "name": "Marc"},
    { "id": "3", "@removed": {} },
    { "id": "4", "name": "Luc"}
  ],
  "@deltaLink": "{opaqueUrl}"
}
```

The results of a request against the delta link may span multiple pages but MUST be ordered by the service across all pages in such a way as to ensure a deterministic result when applied in order to the response that contained the delta link.

If no changes have occurred, the response is an empty collection that contains a delta link for subsequent changes if requested. This delta link MAY be identical to the delta link resulting in the empty collection of changes.

If the delta link is no longer valid, the service MUST respond with*410 Gone*. The response SHOULD include a Location header that the client can use to retrieve a new baseline set of results.

11. JSON standardizations

11.1. JSON formatting standardization for primitive types

Primitive values MUST be serialized to JSON following the rules of [RFC8259][rfc-8259].

Important note for 64bit integers: JavaScript will silently truncate integers larger than `Number.MAX_SAFE_INTEGER` (2^53-1) or numbers smaller than `Number.MIN_SAFE_INTEGER` (-2^53+1). If the service is expected to return integer values outside the range of safe values, strongly consider returning the value as a string in order to maximize interoperability and avoid data loss.

11.2. Guidelines for dates and times

11.2.1. Producing dates

Services MUST produce dates using the `DateLiteral` format, and SHOULD use the `Iso8601Literal` format unless there are compelling reasons to do otherwise. Services that do use the `StructuredDateLiteral` format MUST NOT produce dates using the `T` kind unless BOTH the additional precision is REQUIRED, and ECMAScript clients are explicitly unsupported. (Non-Normative statement: When deciding which particular `DateKind` to standardize on, the approximate order of preference is `E, C, U, W, O, X, I, T`. This optimizes for ECMAScript, .NET, and C++ programmers, in that order.)

11.2.2. Consuming dates

Services MUST accept dates from clients that use the same `DateLiteral` format (including the `DateKind`, if applicable) that they produce, and SHOULD accept dates using any `DateLiteral` format.

11.2.3. Compatibility

Services MUST use the same `DateLiteral` format (including the same `DateKind`, if applicable) for all resources of the same type, and SHOULD use the same `DateLiteral` format (and `DateKind`, if applicable) for all resources across the entire service.

Any change to the `DateLiteral` format produced by the service (including the `DateKind`, if applicable) and any reductions in the `DateLiteral` formats (and `DateKind`, if applicable) accepted by the service MUST be treated as a breaking change. Any widening of the `DateLiteral` formats accepted by the service is NOT considered a breaking change.

11.3. JSON serialization of dates and times

Round-tripping serialized dates with JSON is a hard problem. Although ECMAScript supports literals for most built-in types, it does not define a literal format for dates. The Web has coalesced around the ECMAScript subset of ISO 8601 date formats (ISO 8601), but there are situations where this format is not desirable. For those cases, this document defines a JSON serialization format that can be used to unambiguously represent dates in different formats. Other serialization formats (such as XML) could be derived from this format.

11.3.1. The `DateLiteral` format

Dates represented in JSON are serialized using the following grammar. Informally, a `DateValue` is either an ISO 8601-formatted string or a JSON object containing two properties named `kind` and `value` that together define a point in time. The following is not a context-free grammar; in particular, the interpretation of `DateValue` depends on the value of `DateKind`, but this minimizes the number of productions required to describe the format.

```
DateLiteral:
  Iso8601Literal
  StructuredDateLiteral

Iso8601Literal:
  A string literal as defined in http://www.ecma-international.org/ecma-262/5.1/#sec-15.
  All dates default to UTC unless specified otherwise.

StructuredDateLiteral:
  { DateKindProperty , DateValueProperty }
  { DateValueProperty , DateKindProperty }

DateKindProperty
  "kind" : DateKind

DateKind:
  "C"            ; see below
  "E"            ; see below
  "I"            ; see below
  "O"            ; see below
  "T"            ; see below
  "U"            ; see below
  "W"            ; see below
  "X"            ; see below

DateValueProperty:
  "value" : DateValue

DateValue:
  UnsignedInteger        ; not defined here
  SignedInteger        ; not defined here
  RealNumber        ; not defined here
  Iso8601Literal        ; as above
```

11.3.2. Commentary on date formatting

A `DateLiteral` using the `Iso8601Literal` production is relatively straightforward. Here is an example of an object with a property named `creationDate` that is set to February 13, 2015, at 1:15 p.m. UTC:

```
{ "creationDate" : "2015-02-13T13:15Z" }
```

The `StructuredDateLiteral` consists of a `DateKind` and an accompanying `DateValue` whose valid values (and their interpretation) depend on the `DateKind`. The following table describes the valid combinations and their meaning:

DateKind	DateValue	Colloquial Name & Interpretation	More Info
C	UnsignedInteger	"CLR"; number of milliseconds since midnight January 1, 0001; negative values are not allowed. *See note below.*	MSDN
E	SignedInteger	"ECMAScript"; number of milliseconds since midnight, January 1, 1970.	ECMA International
I	Iso8601Literal	"ISO 8601"; a string limited to the ECMAScript subset.	
O	RealNumber	"OLE Date"; integral part is the number of days since midnight, December 31, 1899, and fractional part is the time within the day (0.5 = midday).	MSDN
T	SignedInteger	"Ticks"; number of ticks (100-nanosecond intervals) since midnight January 1, 1601. *See note below.*	MSDN
U	SignedInteger	"UNIX"; number of seconds since midnight, January 1, 1970.	MSDN
W	SignedInteger	"Windows"; number of milliseconds since midnight January 1, 1601. *See note below.*	MSDN
X	RealNumber	"Excel"; as for `O` but the year 1900 is incorrectly treated as a leap year, and day 0 is "January 0 (zero)."	Microsoft Support

Important note for `C` and `W` kinds: The native CLR and Windows times are represented by 100-nanosecond "tick" values. To interoperate with ECMAScript clients that have limited precision, *these values MUST be converted to and from milliseconds* when (de)serialized as a `DateLiteral`. One millisecond is equivalent to 10,000 ticks.

Important note for `T` kind: This kind preserves the full fidelity of the Windows native time formats (and is trivially convertible to and from the native CLR format) but is incompatible with ECMAScript clients. Therefore, its use SHOULD be limited to only those scenarios that both require the additional precision and do not need to interoperate with ECMAScript clients.

Here is the same example of an object with a property named creationDate that is set to February 13, 2015, at 1:15 p.m. UTC, using several formats:

```
[
  { "creationDate" : { "kind" : "O", "value" : 42048.55 } },
  { "creationDate" : { "kind" : "E", "value" : 1423862100000 } }
]
```

One of the benefits of separating the kind from the value is that once a client knows the kind used by a particular service, it can interpret the value without requiring any additional parsing. In the common case of the value being a number, this makes coding easier for developers:

```
 // We know this service always gives out ECMAScript-format dates
var date = new Date(serverResponse.someObject.creationDate.value);
```

11.4. Durations

Durations need to be serialized in conformance withISO 8601. Durations are "represented by the format `P[n]Y[n]M[n]DT[n]H[n]M[n]S`." From the standard: - P is the duration designator (historically called "period") placed at the start of the duration representation. - Y is the year designator that follows the value for the number of years. - M is the month designator that follows the value for the number of months. - W is the week designator that follows the value for the number of weeks. - D is the day designator that follows the value for the number of days. - T is the time designator that precedes the time components of the representation. - H is the hour designator that follows the value for the number of hours. - M is the minute designator that follows the value for the number of minutes. - S is the second designator that follows the value for the number of seconds.

For example, "P3Y6M4DT12H30M5S" represents a duration of "three years, six months, four days, twelve hours, thirty minutes, and five seconds."

11.5. Intervals

Intervals are defined as part of ISO 8601. - Start and end, such as "2007-03-01T13:00:00Z/2008-05-11T15:30:00Z" - Start and duration, such as "2007-03-01T13:00:00Z/P1Y2M10DT2H30M" - Duration and end, such as "P1Y2M10DT2H30M/2008-05-11T15:30:00Z" - Duration only, such as "P1Y2M10DT2H30M," with additional context information

11.6. Repeating intervals

Repeating Intervals, as per ISO 8601, are:

> Formed by adding "R[n]/" to the beginning of an interval expression, where R is used as the letter itself and [n] is replaced by the number of repetitions. Leaving out the value for [n] means an unbounded number of repetitions.

For example, to repeat the interval of "P1Y2M10DT2H30M" five times starting at "2008-03-01T13:00:00Z," use "R5/2008-03-01T13:00:00Z/P1Y2M10DT2H30M."

12. Versioning

All APIs compliant with the Microsoft REST API Guidelines MUST support explicit versioning. It's critical that clients can count on services to be stable over time, and it's critical that services can add features and make changes.

12.1. Versioning formats

Services are versioned using a Major.Minor versioning scheme. Services MAY opt for a "Major" only version scheme in which case the ".0" is implied and all other rules in this section apply. Two options for specifying the version of a REST API request are supported: - Embedded in the path of the request URL, at the end of the service root: `https://api.contoso.com/v1.0/products/users` - As a query string parameter of the URL: `https://api.contoso.com/products/users?api-version=1.0`

Guidance for choosing between the two options is as follows:

1. Services co-located behind a DNS endpoint MUST use the same versioning mechanism.
2. In this scenario, a consistent user experience across the endpoint is paramount. The Microsoft REST API Guidelines Working Group recommends that new top-level DNS endpoints are not created without explicit conversations with your organization's leadership team.
3. Services that guarantee the stability of their REST API's URL paths, even through future versions of the API, MAY adopt the query string parameter mechanism. This means the naming and structure of the relationships described in the API cannot evolve after the API ships, even across versions with breaking changes.
4. Services that cannot ensure URL path stability across future versions MUST embed the version in the URL path.

Certain bedrock services such as Microsoft's Azure Active Directory may be exposed behind multiple endpoints. Such services MUST support the versioning mechanisms of each endpoint, even if that means supporting multiple versioning mechanisms.

12.1.1. Group versioning

Group versioning is an OPTIONAL feature that MAY be offered on services using the query string parameter mechanism. Group versions allow for logical grouping of API endpoints under a common versioning moniker. This allows developers to look up a single version number and use it across multiple endpoints. Group version numbers are well known, and services SHOULD reject any unrecognized values.

Internally, services will take a Group Version and map it to the appropriate Major.Minor version.

The Group Version format is defined as YYYY-MM-DD, for example 2012-12-07 for December 7, 2012. This Date versioning format applies only to Group Versions and SHOULD NOT be used as an alternative to Major.Minor versioning.

Examples of group versioning

Group	Major.Minor
2012-12-01	1.0
	1.1
	1.2
2013-03-21	1.0
	2.0
	3.0

Group	Major.Minor
	3.1
	3.2
	3.3

Version Format	Example	Interpretation
{groupVersion}	2013-03-21, 2012-12-01	3.3, 1.2
{majorVersion}	3	3.0
{majorVersion}.{minorVersion}	1.2	1.2

Clients can specify either the group version or the Major.Minor version:

For example:

```
GET http://api.contoso.com/acct1/c1/blob2?api-version=1.0
```

```
PUT http://api.contoso.com/acct1/c1/b2?api-version=2011-12-07
```

12.2. When to version

Services MUST increment their version number in response to any breaking API change. See the following section for a detailed discussion of what constitutes a breaking change. Services MAY increment their version number for nonbreaking changes as well, if desired.

Use a new major version number to signal that support for existing clients will be deprecated in the future. When introducing a new major version, services MUST provide a clear upgrade path for existing clients and develop a plan for deprecation that is consistent with their business group's policies. Services SHOULD use a new minor version number for all other changes.

Online documentation of versioned services MUST indicate the current support status of each previous API version and provide a path to the latest version.

12.3. Definition of a breaking change

Changes to the contract of an API are considered a breaking change. Changes that impact the backwards compatibility of an API are a breaking change.

Teams MAY define backwards compatibility as their business needs require. For example, Azure defines the addition of a new JSON field in a response to be not backwards compatible. Office 365 has a looser definition of backwards compatibility and allows JSON fields to be added to responses.

Clear examples of breaking changes:

1. Removing or renaming APIs or API parameters
2. Changes in behavior for an existing API
3. Changes in Error Codes and Fault Contracts
4. Anything that would violate the Principle of Least Astonishment

Services MUST explicitly define their definition of a breaking change, especially with regard to adding new fields to JSON responses and adding new API arguments with default fields. Services that are co-located behind a DNS Endpoint with other services MUST be consistent in defining contract extensibility.

The applicable changes described in this section of the OData V4 spec SHOULD be considered part of the minimum bar that all services MUST consider a breaking change.

13. Long running operations

Long running operations, sometimes called async operations, tend to mean different things to different people. This section sets forth guidance around different types of long running operations, and describes the wire protocols and best practices for these types of operations.

1. One or more clients MUST be able to monitor and operate on the same resource at the same time.
2. The state of the system SHOULD be discoverable and testable at all times. Clients SHOULD be able to determine the system state even if the operation tracking resource is no longer active. The act of querying the state of a long running operation should itself leverage principles of the web. i.e. well-defined resources with uniform interface semantics. Clients MAY issue a GET on some resource to determine the state of a long running operation
3. Long running operations SHOULD work for clients looking to "Fire and Forget" and for clients looking to actively monitor and act upon results.
4. Cancellation does not explicitly mean rollback. On a per-API defined case it may mean rollback, or compensation, or completion, or partial completion, etc. Following a cancelled operation, It SHOULD NOT be a client's responsibility to return the service to a consistent state which allows continued service.

13.1. Resource based long running operations (RELO)

Resource based modeling is where the status of an operation is encoded in the resource and the wire protocol used is the standard synchronous protocol. In this model state transitions are well defined and goal states are similarly defined.

This is the preferred model for long running operations and should be used wherever possible. Avoiding the complexity and mechanics of the LRO Wire Protocol makes things simpler for our users and tooling chain.

An example may be a machine reboot, where the operation itself completes synchronously but the GET operation on the virtual machine resource would have a "state: Rebooting", "state: Running" that could be queried at any time.

This model MAY integrate Push Notifications.

While most operations are likely to be POST semantics, in addition to POST semantics, services MAY support

PUT semantics via routing to simplify their APIs. For example, a user that wants to create a database named "db1" could call:

```
PUT https://api.contoso.com/v1.0/databases/db1
```

In this scenario the databases segment is processing the PUT operation.

Services MAY also use the hybrid defined below.

13.2. Stepwise long running operations

A stepwise operation is one that takes a long, and often unpredictable, length of time to complete, and doesn't offer state transition modeled in the resource. This section outlines the approach that services should use to expose such long running operations.

Service MAY expose stepwise operations.

> Stepwise Long Running Operations are sometimes called "Async" operations. This causes confusion, as it mixes elements of platforms ("Async / await", "promises", "futures") with elements of API operation. This document uses the term "Stepwise Long Running Operation" or often just "Stepwise Operation" to avoid confusion over the word "Async".

Services MUST perform as much synchronous validation as practical on stepwise requests. Services MUST prioritize returning errors in a synchronous way, with the goal of having only "Valid" operations processed using the long running operation wire protocol.

For an API that's defined as a Stepwise Long Running Operation the service MUST go through the Stepwise Long Running Operation flow even if the operation can be completed immediately. In other words, APIs must adopt and stick with an LRO pattern and not change patterns based on circumstance.

13.2.1. PUT

Services MAY enable PUT requests for entity creation.

```
PUT https://api.contoso.com/v1.0/databases/db1
```

In this scenario the *databases* segment is processing the PUT operation.

```
 HTTP/1.1 202 Accepted
Operation-Location: https://api.contoso.com/v1.0/operations/123
```

For services that need to return a 201 Created here, use the hybrid flow described below.

The 202 Accepted should return no body. The 201 Created case should return the body of the target resource.

13.2.2. POST

Services MAY enable POST requests for entity creation.

```
POST https://api.contoso.com/v1.0/databases/

{
  "fileName": "someFile.db",
  "color": "red"
}
```

```
HTTP/1.1 202 Accepted
Operation-Location: https://api.contoso.com/v1.0/operations/123
```

13.2.3. POST, hybrid model

Services MAY respond synchronously to POST requests to collections that create a resource even if the resources aren't fully created when the response is generated. In order to use this pattern, the response MUST include a representation of the incomplete resource and an indication that it is incomplete.

For example:

```
POST https://api.contoso.com/v1.0/databases/ HTTP/1.1
Host: api.contoso.com
Content-Type: application/json
Accept: application/json

{
  "fileName": "someFile.db",
  "color": "red"
}
```

Service response says the database has been created, but indicates the request is not completed by including the Operation-Location header. In this case the status property in the response payload also indicates the operation has not fully completed.

```
HTTP/1.1 201 Created
Location: https://api.contoso.com/v1.0/databases/db1
Operation-Location: https://api.contoso.com/v1.0/operations/123

{
  "databaseName": "db1",
  "color": "red",
  "Status": "Provisioning",
  [ … other fields for "database" …]
}
```

13.2.4. Operations resource

Services MAY provide a "/operations" resource at the tenant level.

Services that provide the "/operations" resource MUST provide GET semantics. GET MUST enumerate the set of operations, following standard pagination, sorting, and filtering semantics. The default sort order for this operation MUST be:

Primary Sort	Secondary Sort
Not Started Operations	Operation Creation Time
Running Operations	Operation Creation Time
Completed Operations	Operation Creation Time

Note that "Completed Operations" is a goal state (see below), and may actually be any of several different states such as "successful", "cancelled", "failed" and so forth.

13.2.5. Operation resource

An operation is a user addressable resource that tracks a stepwise long running operation. Operations MUST support GET semantics. The GET operation against an operation MUST return:

1. The operation resource, it's state, and any extended state relevant to the particular API.
2. 200 OK as the response code.

Services MAY support operation cancellation by exposing DELETE on the operation. If supported DELETE operations MUST be idempotent.

> Note: From an API design perspective, cancellation does not explicitly mean rollback. On a per-API defined case it may mean rollback, or compensation, or completion, or partial completion, etc. Following a cancelled operation, It SHOULD NOT be a client's responsibility to return the service to a consistent state which allows continued service.

Services that do not support operation cancellation MUST return a 405 Method Not Allowed in the event of a DELETE.

Operations MUST support the following states:

1. NotStarted
2. Running
3. Succeeded. Terminal State.
4. Failed. Terminal State.

Services MAY add additional states, such as "Cancelled" or "Partially Completed". Services that support cancellation MUST sufficiently describe their cancellation such that the state of the system can be accurately determined, and any compensating actions may be run.

Services that support additional states should consider this list of canonical names and avoid creating new names if possible: Cancelling, Cancelled, Aborting, Aborted, Tombstone, Deleting, Deleted.

An operation MUST contain, and provide in the GET response, the following information:

1. The timestamp when the operation was created.
2. A timestamp for when the current state was entered.
3. The operation state (notstarted / running / completed).

Services MAY add additional, API specific, fields into the operation. The operation status JSON returned looks like:

```
{
  "createdDateTime": "2015-06-19T12-01-03.45Z",
  "lastActionDateTime": "2015-06-19T12-01-03.45Z",
  "status": "notstarted | running | succeeded | failed"
}
```

Percent complete

Sometimes it is impossible for services to know with any accuracy when an operation will complete. Which makes using the Retry-After header problematic. In that case, services MAY include, in the operationStatus JSON, a percent complete field.

```
{
  "createdDateTime": "2015-06-19T12-01-03.45Z",
  "percentComplete": "50",
  "status": "running"
}
```

In this example the server has indicated to the client that the long running operation is 50% complete.

Target resource location

For operations that result in, or manipulate, a resource the service MUST include the target resource location in the status upon operation completion.

```
{
  "createdDateTime": "2015-06-19T12-01-03.45Z",
  "lastActionDateTime": "2015-06-19T12-06-03.0024Z",
  "status": "succeeded",
  "resourceLocation": "https://api.contoso.com/v1.0/databases/db1"
}
```

13.2.6. Operation tombstones

Services MAY choose to support tombstoned operations. Services MAY choose to delete tombstones after a service defined period of time.

13.2.7. The typical flow, polling

- Client invokes a stepwise operation by invoking an action using POST
- The server MUST indicate the request has been started by responding with a 202 Accepted status code. The response SHOULD include the location header containing a URL that the client should poll for the results after waiting the number of seconds specified in the Retry-After header.
- Client polls the location until receiving a 200 response with a terminal operation state.

Example of the typical flow, polling

Client invokes the restart action:

```
POST https://api.contoso.com/v1.0/databases HTTP/1.1
Accept: application/json

{
  "fromFile": "myFile.db",
  "color": "red"
}
```

The server response indicates the request has been created.

```
HTTP/1.1 202 Accepted
Operation-Location: https://api.contoso.com/v1.0/operations/123
```

Client waits for a period of time then invokes another request to try to get the operation status.

```
GET https://api.contoso.com/v1.0/operations/123
Accept: application/json
```

Server responds that results are still not ready and optionally provides a recommendation to wait 30 seconds.

```
HTTP/1.1 200 OK
Retry-After: 30

{
  "createdDateTime": "2015-06-19T12-01-03.4Z",
  "status": "running"
}
```

Client waits the recommended 30 seconds and then invokes another request to get the results of the operation.

```
GET https://api.contoso.com/v1.0/operations/123
Accept: application/json
```

Server responds with a "status:succeeded" operation that includes the resource location.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "createdDateTime": "2015-06-19T12-01-03.45Z",
  "lastActionDateTime": "2015-06-19T12-06-03.0024Z",
  "status": "succeeded",
  "resourceLocation": "https://api.contoso.com/v1.0/databases/db1"
}
```

13.2.8. The typical flow, push notifications

1. Client invokes a long running operation by invoking an action using POST. The client has a push notification already setup on the parent resource.
2. The service indicates the request has been started by responding with a 202 Accepted status code. The client ignores everything else.
3. Upon completion of the overall operation the service pushes a notification via the subscription on the parent resource.
4. The client retrieves the operation result via the resource URL.

Example of the typical flow, push notifications existing subscription

Client invokes the backup action. The client already has a push notification subscription setup for db1.

```
POST https://api.contoso.com/v1.0/databases/db1?backup HTTP/1.1
Accept: application/json
```

The server response indicates the request has been accepted.

```
HTTP/1.1 202 Accepted
Operation-Location: https://api.contoso.com/v1.0/operations/123
```

The caller ignores all the headers in the return.

The target URL receives a push notification when the operation is complete.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "value": [
    {
      "subscriptionId": "1234-5678-1111-2222",
      "context": "subscription context that was specified at setup",
      "resourceUrl": "https://api.contoso.com/v1.0/databases/db1",
      "userId" : "contoso.com/user@contoso.com",
      "tenantId" : "contoso.com"
    }
  ]
}
```

13.2.9. Retry-After

In the examples above the Retry-After header indicates the number of seconds that the client should wait before trying to get the result from the URL identified by the location header.

The HTTP specification allows the Retry-After header to alternatively specify a HTTP date, so clients should be prepared to handle this as well.

```
HTTP/1.1 202 Accepted
Operation-Location: http://api.contoso.com/v1.0/operations/123
Retry-After: 60
```

Note: The use of the HTTP Date is inconsistent with the use of ISO 8601 Date Format used throughout this document, but is explicitly defined by the HTTP standard in [RFC 7231][rfc-7231-7-1-1-1]. Services SHOULD prefer the integer number of seconds (in decimal) format over the HTTP date format.

13.3. Retention policy for operation results

In some situations, the result of a long running operation is not a resource that can be addressed. For example, if you invoke a long running Action that returns a Boolean (rather than a resource). In these situations, the Location header points to a place where the Boolean result can be retrieved.

Which begs the question: "How long should operation results be retained?"

A recommended minimum retention time is 24 hours.

Operations SHOULD transition to "tombstone" for an additional period of time prior to being purged from the system.

14. Throttling, Quotas, and Limits

14.1. Principles

Services should be as responsive as possible, so as not to block callers. As a rule of thumb any API call that is expected to take longer than 0.5 seconds in the 99th percentile, should consider using the Long-running Operations pattern for those calls. Obviously, services cannot guarantee these response times in the face of potentially unlimited load from callers. Services should therefore design and document call request limits for clients, and respond with appropriate, actionable errors and error messages if these limits are exceeded. Services should respond quickly with an error when they are generally overloaded, rather than simply respond slowly. Finally, many services will have quotas on calls, perhaps a number of operations per hour or day, usually related to a service plan or price. When these quotas are exceeded services must also provide immediate, actionable errors. Quotas and Limits should be scoped to a customer unit: a subscription, a tenant, an application, a plan, or without any other identification a range of ip addresses…as appropriate to the service goals so that the load is properly shared and one unit is not interfering with another.

14.2. Return Codes (429 vs 503)

HTTP specifies two return codes for these scenarios: '429 Too Many Requests' and '503 Service Unavailable'. Services should use 429 for cases where clients are making too many calls and can fix the situation by changing their call pattern. Services should respond with 503 in cases where general load or other problems outside the control of the individual callers is responsible for the service becoming slow. In all cases, services should also provide information suggesting how long the callers should wait before trying in again. Clients should respect these headers and also implement other transient fault handling techniques. However, there may be clients that simply retry immediately upon failure, potentially increasing the load on the service. To handle this, services should design so that returning 429 or 503 is as inexpensive as possible, either by putting in special fastpath code, or ideally by depending on a common frontdoor or load balancer that provides this functionality.

14.3. Retry-After and RateLimit Headers

The Retry-After header is the standard way for responding to clients who are being throttled. It is also common, but optional, in the case of limits and quotas (but not overall system load) to respond with header describing the limit that was exceeded. However, services across Microsoft and the industry use a wide range of different headers for this purpose. We recommend using three headers to describe the limit, the number of calls remaining under the limit, and the time when the limit will reset. However, other headers may be appropriate for specific types of limits. In all cases these must be documented.

14.4. Service Guidance

Services should choose time windows as appropriate for the SLAs or business objectives. In the case of Quotas, the Retry-After time and time window may be very long (hours, days, weeks, even months. Services use 429 to indicate the specific caller has made too many calls, and 503 to indicate that the service is load shedding but that it is not the caller’s responsibility.

14.4.1. Responsiveness

1. Services MUST respond quickly in all circumstances, even when under load.
2. Calls that take longer than 1s to respond in the 99th percentile SHOULD use the Long-Running Operation pattern

3. Calls that take longer than 0.5s to respond in the 99th percentile should strongly consider the LRO pattern
4. Services SHOULD NOT introduce sleeps, pauses, etc. that block callers or are not actionable (“tar-pitting”).

14.4.2. Rate Limits and Quotas

When a caller has made too many calls

1. Services MUST return a 429 code
2. Services MUST return a standard error response describing the specifics so that a programmer can make appropriate changes
3. Services MUST return a Retry-After header that indicates how long clients should wait before retrying
4. Services MAY return RateLimit headers that document the limit or quota that has been exceeded
5. Services MAY return RateLimit-Limit: the number of calls the client is allowed to make in a time window
6. Services MAY return RateLimit-Remaining: the number of calls remaining in the time window
7. Services MAY return RateLimit-Reset: the time at which the window resets in UTC epoch seconds
8. Services MAY return other service specific RateLimit headers as appropriate for more detailed information or specific limits or quotas

14.4.3. Overloaded services

When services are generally overloaded and load shedding

1. Services MUST Return a 503 code
2. Services MUST Return a standard error response (see 7.10.2) describing the specifics so that a programmer can make appropriate changes
3. Services MUST Return a Retry-After header that indicates how long clients should wait before retrying
4. In the 503 case, the service SHOULD NOT return RateLimit headers

14.4.4. Example Response

```
 HTTP/1.1 429 Too Many Requests
Content-Type: application/json
Retry-After: 5
RateLimit-Limit: 1000
RateLimit-Remaining: 0
RateLimit-Reset: 1538152773
{
  "error": {
    "code": "requestLimitExceeded",
    "message": "The caller has made too many requests in the time period.",
    "details": {
      "code": "RateLimit",
       "limit": "1000",
       "remaining": "0",
       "reset": "1538152773",
      }
    }
}
```

14.5. Caller Guidance

Callers include all users of the API: tools, portals, other services, not just user clients

1. Callers MUST wait for a minimum of time indicated in a response with a Retry-After before retrying a request.
2. Callers MAY assume that request is retriable after receiving a response with a Retry-After header without making any changes to the request.
3. Clients SHOULD use shared SDKs and common transient fault libraries to implement the proper behavior

See: https://docs.microsoft.com/en-us/azure/architecture/best-practices/transient-faults

14.6. Handling callers that ignore Retry-After headers

Ideally, 429 and 503 returns are so low cost that even clients that retry immediately can be handled. In these cases, if possible the service team should make an effort to contact or fix the client. If it is a known partner, a bug or incident should be filed. In extreme cases it may be necessary to use DoS style protections such as blocking the caller.

15. Push notifications via webhooks

15.1. Scope

Services MAY implement push notifications via web hooks. This section addresses the following key scenario:

> Push notification via HTTP Callbacks, often called Web Hooks, to publicly-addressable servers.

The approach set forth is chosen due to its simplicity, broad applicability, and low barrier to entry for service subscribers. It's intended as a minimal set of requirements and as a starting point for additional functionality.

15.2. Principles

The core principles for services that support web hooks are:

1. Services MUST implement at least a poke/pull model. In the poke/pull model, a notification is sent to a client, and clients then send a request to get the current state or the record of change since their last notification. This approach avoids complexities around message ordering, missed messages, and change sets. Services MAY add more data to provide rich notifications.
2. Services MUST implement the challenge/response protocol for configuring callback URLs.
3. Services SHOULD have a recommended age-out period, with flexibility for services to vary based on scenario.
4. Services SHOULD allow subscriptions that are raising successful notifications to live forever and SHOULD be tolerant of reasonable outage periods.
5. Firehose subscriptions MUST be delivered only over HTTPS. Services SHOULD require other subscription types to be HTTPS. See the "Security" section for more details.

15.3. Types of subscriptions

There are two subscription types, and services MAY implement either, both, or none. The supported subscription types are:

1. Firehose subscriptions – a subscription is manually created for the subscribing application, typically in an app registration portal. Notifications of activity that any users have consented to the app receiving are sent to this single subscription.
2. Per-resource subscriptions – the subscribing application uses code to programmatically create a subscription at runtime for some user-specific entity(s).

Services that support both subscription types SHOULD provide differentiated developer experiences for the two types:

1. Firehose – Services MUST NOT require developers to create code except to directly verify and respond to notifications. Services MUST provide administrative UI for subscription management. Services SHOULD NOT assume that end users are aware of the subscription, only the subscribing application's functionality.
2. Per-user – Services MUST provide an API for developers to create and manage subscriptions as part of their app as well as verifying and responding to notifications. Services MAY expect end users to be aware of subscriptions and MUST allow end users to revoke subscriptions where they were created directly in response to user actions.

15.4. Call sequences

The call sequence for a firehose subscription MUST follow the diagram below. It shows manual registration of application and subscription, and then the end user making use of one of the service's APIs. At this part of the flow, two things MUST be stored:

1. The service MUST store the end user's act of consent to receiving notifications from this specific application (typically a background usage OAUTH scope.)
2. The subscribing application MUST store the end user's tokens in order to call back for details once notified of changes.

The final part of the sequence is the notification flow itself.

Non-normative implementation guidance: A resource in the service changes and the service needs to run the following logic:

1. Determine the set of users who have access to the resource, and could thus expect apps to receive notifications about it on their behalf.
2. See which of those users have consented to receiving notifications and from which apps.
3. See which apps have registered a firehose subscription.
4. Join 1, 2, 3 to produce the concrete set of notifications that must be sent to apps.

It should be noted that the act of user consent and the act of setting up a firehose subscription could arrive in either order. Services SHOULD send notifications with setup processed in either order.

For a per-user subscription, app registration is either manual or automated. The call flow for a per-user subscription MUST follow the diagram below. It shows the end user making use of one of the service's APIs, and again, the same two things MUST be stored:

1. The service MUST store the end user's act of consent to receiving notifications from this specific application (typically a background usage OAUTH scope).
2. The subscribing application MUST store the end user's tokens in order to call back for details once notified of changes.

In this case, the subscription is set up programmatically using the end-user's token from the subscribing application. The app MUST store the ID of the registered subscription alongside the user tokens.

Non normative implementation guidance: In the final part of the sequence, when an item of data in the service changes and the service needs to run the following logic:

1. Find the set of subscriptions that correspond via resource to the data that changed.
2. For subscriptions created under an app+user token, send a notification to the app per subscription with the subscription ID and user id of the subscription-creator.

- For subscriptions created with an app only token, check that the owner of the changed data or any user that has visibility of the changed data has consented to notifications to the application, and if so send a set of notifications per user id to the app per subscription with the subscription ID.

15.5. Verifying subscriptions

When subscriptions change either programmatically or in response to change via administrative UI portals, the subscribing service needs to be protected from malicious or unexpected calls from services pushing potentially large volumes of notification traffic.

For all subscriptions, whether firehose or per-user, services MUST send a verification request as part of creation or modification via portal UI or API request, before sending any other notifications.

Verification requests MUST be of the following format as an HTTP/HTTPS POST to the subscription's *notificationUrl*.

```
 POST https://{notificationUrl}?validationToken={randomString}
ClientState: clientOriginatedOpaqueToken (if provided by client on subscription-creatior
Content-Length: 0
```

For the subscription to be set up, the application MUST respond with 200 OK to this request, with the *validationToken* value as the sole entity body. Note that if the*notificationUrl* contains query parameters, the *validationToken* parameter must be appended with an `&`.

If any challenge request does not receive the prescribed response within 5 seconds of sending the request, the service MUST return an error, MUST NOT create the subscription, and MUST NOT send further requests or notifications to *notificationUrl*.

Services MAY perform additional validations on URL ownership.

15.6. Receiving notifications

Services SHOULD send notifications in response to service data changes that do not include details of the changes themselves, but include enough information for the subscribing application to respond appropriately to the following process:

1. Applications MUST identify the correct cached OAuth token to use for a callback
2. Applications MAY look up any previous delta token for the relevant scope of change
3. Applications MUST determine the URL to call to perform the relevant query for the new state of the service, which MAY be a delta query.

Services that are providing notifications that will be relayed to end users MAY choose to add more detail to notification packets in order to reduce incoming call load on their service. Such services MUST be clear that notifications are not guaranteed to be delivered and may be lossy or out of order.

Notifications MAY be aggregated and sent in batches. Applications MUST be prepared to receive multiple events inside a single push notification.

The service MUST send all Web Hook data notifications as POST requests.

Services MUST allow for a 30-second timeout for notifications. If a timeout occurs or the application responds with a 5xx response, then the service SHOULD retry the notification with exponential back-off. All other responses will be ignored.

The service MUST NOT follow 301/302 redirect requests.

15.6.1. Notification payload

The basic format for notification payloads is a list of events, each containing the id of the subscription whose referenced resources have changed, the type of change, the resource that should be consumed to identify the exact details of the change and sufficient identity information to look up the token required to call that resource.

For a firehose subscription, a concrete example of this may look like:

```
{
  "value": [
    {
      "subscriptionId": "32b8cbd6174ab18b",
      "resource": "https://api.contoso.com/v1.0/users/user@contoso.com/files?$delta",
      "userId" : "<User GUID>",
      "tenantId" : "<Tenant Id>"
    }
  ]
}
```

For a per-user subscription, a concrete example of this may look like:

```
{
  "value": [
    {
      "subscriptionId": "32b8cbd6174ab183",
      "clientState": "clientOriginatedOpaqueToken",
      "expirationDateTime": "2016-02-04T11:23Z",
      "resource": "https://api.contoso.com/v1.0/users/user@contoso.com/files/$delta",
      "userId" : "<User GUID>",
      "tenantId" : "<Tenant Id>"
    },
    {
      "subscriptionId": "97b391179fa22",
      "clientState ": "clientOriginatedOpaqueToken",
      "expirationDateTime": "2016-02-04T11:23Z",
      "resource": "https://api.contoso.com/v1.0/users/user@contoso.com/files/$delta",
      "userId" : "<User GUID>",
      "tenantId" : "<Tenant Id>"
    }
  ]
}
```

Following is a detailed description of the JSON payload.

A notification item consists a top-level object that contains an array of events, each of which identified the subscription due to which this notification is being sent.

Field	Description
value	Array of events that have been raised within the subscription's scope since the last notification.

Each item of the events array contains the following properties:

Field	Description
subscriptionId	The id of the subscription due to which this notification has been sent. Services MUST provide the *subscriptionId* field.
clientState	Services MUST provide the *clientState* field if it was provided at subscription creation time.
expirationDateTime	Services MUST provide the *expirationDateTime* field if the subscription has one.
resource	Services MUST provide the resource field. This URL MUST be considered opaque by the subscribing application. In the case of a richer notification it MAY be subsumed by message content that implicitly contains the resource URL to avoid duplication. If a service is providing this data as part of a more detailed data packet, then it need not be duplicated.
userId	Services MUST provide this field for user-scoped resources. In the case of user-scoped resources, the unique identifier for the user should be used. In the case of resources shared between a specific set of users, multiple notifications must be sent, passing the unique identifier of each user. For tenant-scoped resources, the user id of the subscription should be used.
tenantId	Services that wish to support cross-tenant requests SHOULD provide this field. Services that provide notifications on tenant-scoped data MUST send this field.

15.7. Managing subscriptions programmatically

For per-user subscriptions, an API MUST be provided to create and manage subscriptions. The API must support at least the operations described here.

15.7.1. Creating subscriptions

A client creates a subscription by issuing a POST request against the subscriptions resource. The subscription namespace is client-defined via the POST operation.

```
https://api.contoso.com/apiVersion/$subscriptions
```

The POST request contains a single subscription object to be created. That subscription object has the following properties:

Property Name	Required	Notes

Property Name	Required	Notes
resource	Yes	Resource path to watch.
notificationUrl	Yes	The target web hook URL.
clientState	No	Opaque string passed back to the client on all notifications. Callers may choose to use this to provide tagging mechanisms.

If the subscription was successfully created, the service MUST respond with the status code 201 CREATED and a body containing at least the following properties:

Property Name	Required	Notes
id	Yes	Unique ID of the new subscription that can be used later to update/delete the subscription.
expirationDateTime	No	Uses existing Microsoft REST API Guidelines defined time formats.

Creation of subscriptions SHOULD be idempotent. The combination of properties scoped to the auth token, provides a uniqueness constraint.

Below is an example request using a User + Application principal to subscribe to notifications from a file:

```
POST https://api.contoso.com/files/v1.0/$subscriptions HTTP 1.1
Authorization: Bearer {UserPrincipalBearerToken}

{
  "resource": "http://api.service.com/v1.0/files/file1.txt",
  "notificationUrl": "https://contoso.com/myCallbacks",
  "clientState": "clientOriginatedOpaqueToken"
}
```

The service SHOULD respond to such a message with a response format minimally like this:

```
{
  "id": "32b8cbd6174ab18b",
  "expirationDateTime": "2016-02-04T11:23Z"
}
```

Below is an example using an Application-Only principal where the application is watching all files to which it's authorized:

```
POST https://api.contoso.com/files/v1.0/$subscriptions HTTP 1.1
Authorization: Bearer {ApplicationPrincipalBearerToken}

{
  "resource": "All.Files",
  "notificationUrl": "https://contoso.com/myCallbacks",
  "clientState": "clientOriginatedOpaqueToken"
}
```

The service SHOULD respond to such a message with a response format minimally like this:

```
{
  "id": "8cbd6174abb391179",
  "expirationDateTime": "2016-02-04T11:23Z"
}
```

15.7.2. Updating subscriptions

Services MAY support amending subscriptions. To update the properties of an existing subscription, clients use PATCH requests providing the ID and the properties that need to change. Omitted properties will retain their values. To delete a property, assign a value of JSON null to it.

As with creation, subscriptions are individually managed.

The following request changes the notification URL of an existing subscription:

```
PATCH https://api.contoso.com/files/v1.0/$subscriptions/{id} HTTP 1.1
Authorization: Bearer {UserPrincipalBearerToken}

{
  "notificationUrl": "https://contoso.com/myNewCallback"
}
```

If the PATCH request contains a new *notificationUrl*, the server MUST perform validation on it as described above. If the new URL fails to validate, the service MUST fail the PATCH request and leave the subscription in its previous state.

The service MUST return an empty body and `204 No Content` to indicate a successful patch.

The service MUST return an error body and status code if the patch failed.

The operation MUST succeed or fail atomically.

15.7.3. Deleting subscriptions

Services MUST support deleting subscriptions. Existing subscriptions can be deleted by making a DELETE request against the subscription resource:

```
 DELETE https://api.contoso.com/files/v1.0/$subscriptions/{id} HTTP 1.1
Authorization: Bearer {UserPrincipalBearerToken}
```

As with update, the service MUST return `204 No Content` for a successful delete, or an error body and status code to indicate failure.

15.7.4. Enumerating subscriptions

To get a list of active subscriptions, clients issue a GET request against the subscriptions resource using a User + Application or Application-Only bearer token:

```
 GET https://api.contoso.com/files/v1.0/$subscriptions HTTP 1.1
Authorization: Bearer {UserPrincipalBearerToken}
```

The service MUST return a format as below using a User + Application principal bearer token:

```
{
  "value": [
    {
      "id": "32b8cbd6174ab18b",
      "resource": " http://api.contoso.com/v1.0/files/file1.txt",
      "notificationUrl": "https://contoso.com/myCallbacks",
      "clientState": "clientOriginatedOpaqueToken",
      "expirationDateTime": "2016-02-04T11:23Z"
    }
  ]
}
```

An example that may be returned using Application-Only principal bearer token:

```
{
  "value": [
    {
      "id": "6174ab18bfa22",
      "resource": "All.Files ",
      "notificationUrl": "https://contoso.com/myCallbacks",
      "clientState": "clientOriginatedOpaqueToken",
      "expirationDateTime": "2016-02-04T11:23Z"
    }
  ]
}
```

15.8. Security

All service URLs must be HTTPS (that is, all inbound calls MUST be HTTPS). Services that deal with Web Hooks MUST accept HTTPS.

We recommend that services that allow client defined Web Hook Callback URLs SHOULD NOT transmit data over HTTP. This is because information can be inadvertently exposed via client, network, server logs and other mechanisms.

However, there are scenarios where the above recommendations cannot be followed due to client endpoint or software limitations. Consequently, services MAY allow web hook URLs that are HTTP.

Furthermore, services that allow client defined HTTP web hooks callback URLs SHOULD be compliant with privacy policy specified by engineering leadership. This will typically include recommending that clients prefer SSL connections and adhere to special precautions to ensure that logs and other service data collection are properly handled.

For example, services may not want to require developers to generate certificates to onboard. Services might only enable this on test accounts.

16. Unsupported requests

RESTful API clients MAY request functionality that is currently unsupported. RESTful APIs MUST respond to valid but unsupported requests consistent with this section.

16.1. Essential guidance

RESTful APIs will often choose to limit functionality that can be performed by clients. For instance, auditing systems allow records to be created but not modified or deleted. Similarly, some APIs will expose collections but require or otherwise limit filtering and ordering criteria, or MAY not support client-driven pagination.

16.2. Feature allow list

If a service does not support any of the below API features, then an error response MUST be provided if the feature is requested by a caller. The features are: - Key Addressing in a collection, such as: `https://api.contoso.com/v1.0/people/user1@contoso.com` - Filtering a collection by a property value, such as: `https://api.contoso.com/v1.0/people?$filter=name eq 'david'` - Filtering a collection by range, such as: `http://api.contoso.com/v1.0/people?$filter=hireDate ge 2014-01-01 and hireDate le 2014-12-31` - Client-driven pagination via $top and $skip, such as: `http://api.contoso.com/v1.0/people?$top=5&$skip=2` - Sorting by $orderBy, such as: `https://api.contoso.com/v1.0/people?$orderBy=name desc` - Providing $delta tokens, such as: `https://api.contoso.com/v1.0/people?$delta`

16.2.1. Error response

Services MUST provide an error response if a caller requests an unsupported feature found in the feature allow list. The error response MUST be an HTTP status code from the 4xx series, indicating that the request cannot be fulfilled. Unless a more specific error status is appropriate for the given request, services SHOULD return "400 Bad Request" and an error payload conforming to the error response guidance provided in the Microsoft REST API Guidelines. Services SHOULD include enough detail in the response message for a developer to determine exactly what portion of the request is not supported.

Example:

```
GET https://api.contoso.com/v1.0/people?$orderBy=name HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": {
    "code": "ErrorUnsupportedOrderBy",
    "message": "Ordering by name is not supported."
  }
}
```

17. Naming guidelines

17.1. Approach

Naming policies should aid developers in discovering functionality without having to constantly refer to documentation. Use of common patterns and standard conventions greatly aids developers in correctly guessing common property names and meanings. Services SHOULD use verbose naming patterns and SHOULD NOT use abbreviations other than acronyms that are the dominant mode of expression in the domain being represented by the API, (e.g. Url).

17.2. Casing

- Acronyms SHOULD follow the casing conventions as though they were regular words (e.g. Url).
- All identifiers including namespaces, entityTypes, entitySets, properties, actions, functions and enumeration values SHOULD use lowerCamelCase.
- HTTP headers are the exception and SHOULD use standard HTTP convention of Capitalized-Hyphenated-Terms.

17.3. Names to avoid

Certain names are so overloaded in API domains that they lose all meaning or clash with other common usages in domains that cannot be avoided when using REST APIs, such as OAUTH. Services SHOULD NOT use the following names: - Context - Scope - Resource

17.4. Forming compound names

- Services SHOULD avoid using articles such as 'a', 'the', 'of' unless needed to convey meaning.
  - e.g. names such as aUser, theAccount, countOfBooks SHOULD NOT be used, rather user, account, bookCount SHOULD be preferred.
- Services SHOULD add a type to a property name when not doing so would cause ambiguity about how

the data is represented or would cause the service not to use a common property name.
- When adding a type to a property name, services MUST add the type at the end, e.g. createdDateTime.

17.5. Identity properties

- Services MUST use string types for identity properties.
- For OData services, the service MUST use the OData @id property to represent the canonical identifier of the resource.
- Services MAY use the simple 'id' property to represent a local or legacy primary key value for a resource.
- Services SHOULD use the name of the relationship postfixed with 'Id' to represent a foreign key to another resource, e.g. subscriptionId.
    - The content of this property SHOULD be the canonical ID of the referenced resource.

17.6. Date and time properties

- For properties requiring both date and time, services MUST use the suffix 'DateTime'.
- For properties requiring only date information without specifying time, services MUST use the suffix 'Date', e.g. birthDate.
- For properties requiring only time information without specifying date, services MUST use the suffix 'Time', e.g. appointmentStartTime.

17.7. Name properties

- For the overall name of a resource typically shown to users, services MUST use the property name 'displayName'.
- Services MAY use other common naming properties, e.g. givenName, surname, signInName.

17.8. Collections and counts

- Services MUST name collections as plural nouns or plural noun phrases using correct English.
- Services MAY use simplified English for nouns that have plurals not in common verbal usage.
    - e.g. schemas MAY be used instead of schemata.
- Services MUST name counts of resources with a noun or noun phrase suffixed with 'Count'.

17.9. Common property names

Where services have a property, whose data matches the names below, the service MUST use the name from this table. This table will grow as services add terms that will be more commonly used. Service owners adding such terms SHOULD propose additions to this document.

| | |------------- | attendees | body |
createdDateTime | childCount |
children |
contentUrl |
country |
createdBy |
displayName | errorUrl | eTag | event | expirationDateTime | givenName | jobTitle | kind |

id | lastModifiedDateTime | location | memberOf | message | name |
owner | people |
person |
postalCode | photo | preferredLanguage | properties | signInName | surname | tags | userPrincipalName | webUrl
|

18. Appendix

18.1. Sequence diagram notes

All sequence diagrams in this document are generated using theWebSequenceDiagrams.com. To generate them, paste the text below into the web tool.

18.1.1. Push notifications, per user flow

```
 === Begin Text ===
note over Developer, Automation, App Server:
     An App Developer like MovieMaker
     Wants to integrate with primary service like Dropbox
end note
note over DB Portal, DB App Registration, DB Notifications, DB Auth, DB Service: The pri
note over Client: The end users' browser or installed app

note over Developer, Automation, App Server, DB Portal, DB App Registration, DB Notifica


Developer <--> DB Portal : Login into Portal, App Registration UX
DB Portal -> +DB App Registration: App Name etc.
note over DB App Registration: Confirm Portal Access Token

DB App Registration -> -DB Portal: App ID
DB Portal <--> App Server: Developer copies App ID

note over Developer, Automation, App Server, DB Portal, DB App Registration, DB Notifica

Developer <--> DB Portal: webhook registration UX
DB Portal -> +DB Notifications: Register: App Server webhook URL, Scope, App ID
Note over DB Notifications : Confirm Portal Access Token
DB Notifications -> -DB Portal: notification ID
DB Portal --> App Server : Developer may copy notification ID


note over Developer, Automation, App Server, DB Portal, DB App Registration, DB Notifica

Client -> +App Server : Request access to DB protected information
App Server -> -Client : Redirect to DB Authorization endpoint with authorization request
Client -> +DB Auth : Redirected authorization request
Client <--> DB Auth : Authorization UX
DB Auth -> -Client : Redirect back to App Server with code
Client -> +App Server : Redirect request back to access server with access code
App Server -> +DB Auth : Request tokens with access code
note right of DB Service: Cache that this User ID provided access to App ID
DB Auth -> -App Server : Response with access, refresh, and ID tokens
note right of App Server : Cache tokens by user ID
App Server -> -Client : Return information to client

note over Developer, Automation, App Server, DB Portal, DB App Registration, DB Notifica

Client <--> DB Service: Changes to user data - typical via interacting with App Server v
DB Service -> App Server : Notification with notification ID and user ID
App Server -> +DB Service : Request changed information with cached access tokens and "s
note over DB Service: Confirm User Access Token
DB Service -> -App Server : Response with data and new "since" token
note right of App Server: Update status and cache new "since" token
=== End Text ===
```

18.1.2. Push notifications, firehose flow

```
 === Begin Text ===
note over Developer, Automation, App Server:
     An App Developer like MovieMaker
     Wants to integrate with primary service like Dropbox
end note
note over DB Portal, DB App Registration, DB Notifications, DB Auth, DB Service: The pri
note over Client: The end users' browser or installed app

note over Developer, Automation, App Server, DB Portal, DB App Registration, DB Notifica

alt Automated app registration
   Developer <--> Automation: Configure
   Automation -> +DB App Registration: App Name etc.
   note over DB App Registration: Confirm App Access Token
   DB App Registration -> -Automation: App ID, App Secret
   Automation --> App Server : Embed App ID, App Secret
else Manual app registration
   Developer <--> DB Portal : Login into Portal, App Registration UX
   DB Portal -> +DB App Registration: App Name etc.
   note over DB App Registration: Confirm Portal Access Token

   DB App Registration -> -DB Portal: App ID
   DB Portal <--> App Server: Developer copies App ID
end

note over Developer, Automation, App Server, DB Portal, DB App Registration, DB Notifica
```

```
Client -> +App Server : Request access to DB protected information
App Server -> -Client : Redirect to DB Authorization endpoint with authorization request
Client -> +DB Auth : Redirected authorization request
Client <--> DB Auth : Authorization UX
DB Auth -> -Client : Redirect back to App Server with code
Client -> +App Server : Redirect request back to access server with access code
App Server -> +DB Auth : Request tokens with access code
note right of DB Service: Cache that this User ID provided access to App ID
DB Auth -> -App Server : Response with access, refresh, and ID tokens
note right of App Server : Cache tokens by user ID
App Server -> -Client : Return information to client



note over Developer, Automation, App Server, DB Portal, DB App Registration, DB Notifica

App Server->+DB Notifications: Register: App server webhook URL, Scope, App ID
note over DB Notifications : Confirm User Access Token
DB Notifications -> -App Server: notification ID
note right of App Server : Cache the Notification ID and User Access Token



note over Developer, Automation, App Server, DB Portal, DB App Registration, DB Notifica

Client <--> DB Service: Changes to user data - typical via interacting with App Server v
DB Service -> App Server : Notification with notification ID and user ID
App Server -> +DB Service : Request changed information with cached access tokens and "s
note over DB Service: Confirm User Access Token
DB Service -> -App Server : Response with data and new "since" token
note right of App Server: Update status and cache new "since" token



=== End Text ===
```