OData Version 3.0 Core Protocol

OData Version 4.0 is the current recommended version of OData. OData V4 has been standardized by OASIS and has many features not included in OData Version 3.0.

➜ **Go to OData Version 4.0 (/documentation/)**

1. Overview

The OData Protocol is an application-level protocol for interacting with data via RESTful web services. The protocol supports the description of data models and the editing and querying of data according to those models. It provides facilities for:

- Metadata: a machine-readable description of the data model exposed by a particular data provider.
- Data: sets of data entities and the relationships between them.
- Querying: requesting that the service perform a set of filtering and other transformations to its data, then return the results.
- Editing: creating, editing, and deleting data.
- Operations: invoking custom logic
- Vocabularies: attaching custom semantics

The OData Protocol is different from other REST-based web service approaches in that it provides a uniform way to describe both the data and the data model. This improves semantic interoperability between systems and allows an ecosystem to emerge.

Towards that end, the OData Protocol follows these design principles:

- Prefer mechanisms that work on a variety of data stores. In particular, do not assume a relational data model.
- Backwards compatibility is paramount. Clients and services which speak different versions of the OData Protocol should interoperate, supporting everything allowed in lower versions.
- Follow REST principles unless there is a good and specific reason not to.
- OData should degrade gracefully. It should be easy to build a very basic but compliant OData service, with additional work necessary only to support additional capabilities.
- Keep it simple. Address the common cases and provide extensibility where necessary.

2. Data Model

This section provides a high-level description of the *Entity Data Model (EDM)*: the abstract data model that MUST be used to describe the data exposed by an OData service. An OData Metadata Document is a representation of a service’s data model exposed for client consumption.

The central concepts in the EDM are *entities*, *entity sets*, and *relationships*.

Entities are instances of entity types (e.g. `Customer` , `Employee` , etc.). *Entity types* are nominal structured types with a key. Entities consist of named properties and MAY include relationships with other entities. Entity types support single inheritance from other entity types. Entities are the core identity types in a data model.

Entity sets are named collections of entities (e.g. `Customers` is an entity set containing `Customer` entities). An entity can be a member of at most one entity set. Entity sets provide the primary entry points into the data model.

Entities and entity sets are *relatable types*.

Relationships have a name and are used to navigate from an entity to related entities. Relationships are represented in entity types as *navigation properties*. Relationships may be addressed directly through a *navigation link* representing the relationship itself. Each relationship has a cardinality.

Complex types are keyless nominal structural types consisting of a set of properties. These are value types that lack identity. Complex types are commonly used as property values in an entity or as parameters to operations.

The *entity key* of an entity type is formed from a subset of primitive properties (e.g. `CustomerId`, `OrderId,` LineId`, etc.) of the entity type. The entity key value uniquely identifies an entity within a collection of entities.

Properties declared as part of the entity type's definition are called *declared properties*. Entity types which allow additional undeclared properties are called *open entity types*. These additional properties are called *dynamic properties*. A dynamic property MUST NOT have the same name as a declared property.

Operations allow the execution of custom logic on parts of a data model. *Functions* do not allow side effects and are composable. *Actions* allow side effects and are not composable. Actions and functions are global to the service and MAY be used as members of entities and collections of entities.

Entity sets and operations are grouped in a named *entity container*. This container represents a service's model.

Structural elements are composed of other model elements. Entity types, complex types, association types, and row types are all structural elements. *Row types* are unnamed structural elements.

An OData *resource* is anything in the model that can be addressed (an entity set, entity, property, or operation).

Refer to OData:CSDL (/documentation/odata-version-3-0/common-schema-definition-language-csdl) for more information on the OData entity data model.

2.1 Annotations

Model and instance elements may be annotated with type annotations or value annotations.

Type Annotations are defined in metadata as entity types or complex types, and are generally used to define a common concept, such as a person or a movie.

Value Annotations are defined in metadata as individual value terms with a name and a type. Value annotations are typically used to specify an individual fact about an element, such as whether it is read-only.

A set of related type annotation terms or value annotation terms in a common namespace comprises a *Vocabulary*.

Applied *annotations* have a *term* (the namespace-qualified name of the annotation being applied), a *target* (the model or instance element to which the term is applied), and a *value*. The value may be a static value, or an expression which may contain a path to one or more properties of an annotated entity.

3. Service Model

OData allows generic clients to discover the capabilities of services. This is possible because the service is defined in a uniform way, using a comon data model. The service advertises its concrete data model in a machine-readable form.

An OData service consists of two well-defined, static resources and a set of dynamic resources. The two static resources allow machines to ask a service about its data model. The dynamic resources provide ways to manipulate that model.

The *metadata document* is a static resource that describes the data model and type system understood by that particular OData service.

Clients can use the metadata document to understand how to query and navigate between the entities in the system.

The second static resource in an OData service is the service document. The *service document* lists all of the top-level entity sets exposed by the service.

The rest of an OData service consists of dynamic resources. The URLs for many of these resources can be computed from the information in the metadata document.

A typical OData interaction proceeds as follows:

1. Client has an intent
2. Client asks for and parses the metadata document.
3. Client uses metadata to determine how to form a request for its intent.
4. Client performs request(s) to interact with data.
5. Each response provides additional options to the client, in terms of both links and instance metadata.
6. Client can follow links or combine instance metadata with service metadata to determine new entry points.

In this way, the service always remains in control. It defines what is allowed at any moment and how that will be requested.

However, the service can describe those operations in terms of composable chunks. This allows the client wide flexibility in how it composes resources to achieve its intent.

See Requesting Data and Data Modification for details.

4. Notational Conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC2119 (https://tools.ietf.org/html/rfc2119), “Key words for use in RFCs to Indicate Requirement Levels”)].

4.1. Normative References

This document references the following related documents:

- OData:CSDL (/documentation/odata-version-3-0/common-schema-definition-language-csdl). Detailed description of the OData Entity Data Model.
- OData:URL (/documentation/odata-version-3-0/url-conventions). Conventions for constructing OData requests.

- OData:ABNF (/documentation/odata-version-3-0/abnf) Full ABNF rules for OData requests.
- OData:ATOM (/documentation/odata-version-3-0/atom-format) ATOM encoding for OData payloads. OData Services MUST support the ATOM encoding.
- OData:JSON (/documentation/odata-version-3-0/json-verbose-format) A JSON encoding for OData payloads. OData services SHOULD support a JSON encoding.
- OData:Batch (/documentation/odata-version-3-0/batch-processing) Support for grouping multiple OData requests in a single batch.
- RFC2616 (https://www.w3.org/Protocols/rfc2616/rfc2616.html) HTTP 1.1 Specification
- RFC5023 (https://tools.ietf.org/html/rfc5023) The Atom Publishing Protocol
- RFC2119 (https://tools.ietf.org/html/rfc2119) Keywords for use in RFCs to Indicate Requirement Levels
- RFC5789 (https://tools.ietf.org/html/rfc5789) Patch Method for HTTP

4.2. Example Payloads

Some sections of this specification are illustrated with non-normative example OData request and response payloads. However, the text of this specification provides the definition of conformance.

4.3. Interpreting Examples

All code examples in this document are non-normative.

5. Versioning

This document defines version 3.0 of the OData Specification.

The OData protocol supports a versioning scheme for enabling services to expose new features and format versions without breaking compatibility with older clients.

OData requests and responses MAY be versioned according to The `DataServiceVersion` Header.

An OData client SHOULD use the MinDataServiceVersion and MaxDataServiceVersion headers in order to specify the range of acceptable response DataServiceVersions.

The service SHOULD respond with the maximum version supported by the service that is less than or equal to the specified `MaxDataServiceVersion`.

`DataServiceVersion`, `MinDataServiceVersion`, and `MaxDataServiceVersion` header fields MUST be of the following form:

```
majorversionnumber + "." + minorversionnumber
```

This version of the specification defines the following valid data service version values: “1.0”, “2.0”, and “3.0”, corresponding to OData versions 1.0, 2.0, and 3.0, respectively.

6. Extensibility

The OData protocol supports both user- and version-driven extensibility through a combination of versioning, convention, and explicit extension points.

6.1. Query Option Extensibility

Query options within the request URL can control how a particular request is processed by the service.

OData-defined system query options are prefixed with “$”. Services MAY support additional query options not defined in the OData specification, but they MUST NOT begin with the “$” character.

OData services SHOULD NOT require any query options to be specified in a request, and SHOULD fail any request that contains query options that it does not understand.

6.2. Payload Extensibility

OData supports extensibility in the payload, according to the specific format.

Regardless of the format, additional content MUST NOT be present if it needs to be understood by the receiver in order to correctly interpret the payload. Thus, clients and services MAY safely ignore any content not specifically defined in the version of the payload specified by the [`DataServiceVersion` (#thedataserviceversionheader) header.

6.3. Action/Function Extensibility

Actions and functions extend the set of operations that can be performed on or with a service or resource. Actions MAY have side-effects. For example actions may be used to extend CUD operations or to invoke custom operations. Functions MUST NOT have side-effects. Functions can be invoked:

- directly from the service root
- from an url that addresses a resource
- inside a predicate to a `$filter` or `$orderby` system query option.

Fully qualified action and function names include a namespace prefix. The `odata` and `geo` namespaces are reserved for the use of this specification.

An OData service MUST fail any request that contains actions or functions that it does not understand.

6.4. Vocabulary Extensibility

Vocabularies provide the ability to annotate metadata as well as instance data, and define a powerful extensibility point for OData.

Metadata annotations can be used to define additional characteristics or capabilities of a metadata element, such as a service, entity type, property, function, action or parameter. For example, a metadata annotation may define ranges of valid values for a particular property.

Instance annotations can be used to define additional information associated with a particular result, entity, property, or error; for example whether a property is read-only for a particular instance.

Annotations that apply across instances SHOULD be specified within the metadata. Where the same annotation is defined at both the metadata and instance level, the instance-level annotation SHOULD override the annotation specified at the metadata level.

A service SHOULD NOT require a client to interpret annotations it uses.

6.5. Header Field Extensibility

OData defines semantics around certain HTTP request and response headers. Services that support a version of OData MUST understand and comply with the headers defined by that version. Compliance means either honoring the semantics of the header field or failing the request.

Individual services MAY define custom headers. These headers MUST NOT begin with `OData-` . Custom headers SHOULD be optional when making requests to the service. A service MUST NOT require a client to understand custom headers to accurately interpret the response.

6.6. Format Extensibility

An OData Service MUST support at least the OData:Atom (/documentation/odata-version-3-0/atom-format) format, and MAY support additional formats for both request and response bodies.

7. Formats

The client may request a particular response format through the `Accept` header, as specified in RFC2616 (https://www.w3.org/Protocols/rfc2616/rfc2616.html), or through the `$format` System Query Option.

In the case that both the `Accept` header and the `$format` query option are specified on a request, the value specified in the `$format` query option SHOULD be used.

If the service does not support the requested format, it SHOULD reply with a `406 Not Acceptable` error response.

See the format-specific specifications (OData:JSON (/documentation/odata-version-3-0/json-verbose-format), OData:Atom (/documentation/odata-version-3-0/atom-format)) for details.

8. Header Fields

OData defines semantics around the following request and response headers. Additional headers MAY be specified, but have no unique semantics defined in OData.

8.1. Common Headers

The `DataServiceVersion` and `Content-Type` headers may be used on any OData request or response.

8.1.1. The `DataServiceVersion` Header

OData clients MAY use the `DataServiceVersion` header on a request to specify the version of the protocol used to generate the request.

If present on a request, the service MUST interpret the request according to the rules defined in the specified version of the protocol, or fail the request with a 4xx response code.

If not specified, the service MUST assume the request is generated using the maximum version of the protocol that the service understands.

OData services SHOULD specify the `DataServiceVersion` header on a response to specify the version of the protocol used to generate the response. If present on a response, the client MUST interpret the response according to the rules defined in the specified version of the protocol. If not specified, the client MUST assume the request is generated using version 1.0 of the OData Protocol.

For more details see Versioning.

8.1.2. The `Content-Type` Header

The format of an individual request or response body MUST be specified in the `Content-Type` header of the request or response.

See the format-specific specifications (OData:JSON (/documentation/odata-version-3-0/json-verbose-format), OData:Atom (/documentation/odata-version-3-0/atom-format)) for details.

8.2. Common Request Headers

In addition to the Common Headers, a client MAY specify any combination of the following request headers.

8.2.1. The `MaxDataServiceVersion` Request Header

Clients SHOULD specify a `MaxDataServiceVersion` request header.

If specified, the service MUST generate a response with a `DataServiceVersion` less than or equal to the specified `MaxDataServiceVersion`.

If `MaxDataServiceVersion` is not specified, then the service SHOULD interpret the request as having a `MaxDataServiceVersion` equal to the maximum version supported by the service.

For more details see Versioning.

8.2.2. The `MinDataServiceVersion` Request Header

Clients SHOULD specify a `MinDataServiceVersion` request header.

If specified, the service MUST generate a response with a DataServiceVersion greater than or equal to the specified `MinDataServiceVersion`.

The service SHOULD respond with the maximum version supported by the service that is less than or equal to the specified `MaxDataServiceVersion`.

For more details see Versioning.

8.2.3. The `Accept` Request Header

As specified in RFC2616 (https://www.w3.org/Protocols/rfc2616/rfc2616.html), the client MAY specify the set of accepted formats through the use of the `Accept` Header.

8.2.4. The `If-Match` Request Header

A client MAY include an `If-Match` header in a request to GET, PUT, MERGE, PATCH or DELETE an entity or entity property, or to invoke an action bound to an entity. The value of the `If-Match` request header MUST be an ETag value previously retrieved for the entity.

If specified, the request MUST only be invoked if the specified value matches the current ETag value of the entity. If the value does not match the current ETag value of the entity for a Data Modification or Action request, the service MUST respond with '412 Precondition Failed' and MUST ensure that no data is modified as a result of the request.

8.2.5. The `If-None-Match` Request Header

A client MAY include an `If-None-Match` header in a request to GET, PUT, MERGE, PATCH or DELETE an entity or entity property, or to invoke an action bound to an entity. The value of the `If-None-Match` request header MUST be an ETag value previously retrieved for the entity.

If specified, the request MUST only be invoked if the specified value does not match the current ETag value of the entity. If the value does match the current ETag value of the entity for a Data Modification or Action request, the service MUST respond with '412 Precondition Failed' and MUST ensure that no data is modified as a result of the request.

8.3. Common Response Headers

In addition to the Common Headers, a service MAY specify the following response headers.

8.3.1. The `ETag` Header

A request that returns an individual entity MAY include an `ETag` header.

The value specified in the `ETag` header may be specified in the `If-Match` or `If-None-Match` header of a subsequent Data Modification or Action request in order to apply optimistic concurrency in updating, deleting, or invoking the action bound to, the entity.

8.3.2. The `Location` Header

The Location header is used to specify the URL of an entity modified through a Data Modification request, or the request URL to check on the status of an asynchronous operation as described in `202 Accepted` .

8.4. Data Modification Request Headers

In addition to the Common Headers, a client MAY specify the following headers on a Data Modification request.

8.4.1. The `Prefer` Header

The client MAY specify a `Prefer` header on Data Modification or Action request.

A Prefer header with a value of `return-no-content` requests that the service invoke the request but not return content in the response. The service MAY honor this request by returning `204 No Content` .

A Prefer header with a value of `return-content` requests that the service invoke the request and return the modified entity. The service MAY honor this request by returning the successfully modified entity in the body of the response, formatted according to the rules specified for the requested format.

In response to a request containing a `Prefer` header, the service MAY return the `Preference-Applied` Header.

8.5. Data Modification Response Headers

In addition to the Common Headers, a service MAY specify the following headers on a Data Modification response.

8.5.1. The `DataServiceId` Header

A response to a PUT, POST, MERGE, or PATCH request that returns `404 No Content` MUST include a DataServiceId response header. The value of the header is the URI identifier of the entity that was acted on by the request.

8.5.2. The `Preference-Applied` Header

In response to a Data Modification or Action request containing a `Prefer header` , the service may include a `Preference-Applied` response header to specify the `prefer` header value that was honored.

If the service has returned content in response to a request including a `Prefer` header with a value of `return-content` , it MAY include a `Preference-Applied` response header with a value of `return-content` .

If the service has returned content in response to a request including a `Prefer` header with a value of `return-content` , it MAY include a `Preference-Applied` response header with a value of `return-no-content` .

8.5.3. The `Retry-After` Header

A service MUST include a `Retry-After` header in a `202 Accepted` response.

The Retry-After header specifies the suggested length of time, in seconds, after which the client SHOULD use the URL returned in the `Location Header` to check on the status of the operation.

9. Common Response Semantics

An OData service MAY respond to any request using any valid HTTP status code appropriate for the request. A service SHOULD be as specific as possible in its choice of HTTP status codes.

The following represent the most common success response codes. In some cases, a service MAY respond with a more specific success code.

9.1. Success Response Codes

The following response codes represent successful requests.

9.1.1. `200 OK` Response Code

A GET, PUT, MERGE, or PATCH request MAY return `200 OK` if the operation is completed successfully. In this case, the response body MUST contain the value of the entity or property specified in the request URL.

9.1.2. `201 Created` Response Code

A POST request MAY return `201 Created` if the entity or link was successfully created. In this case, the response body MUST contain the updated entity.

9.1.3. `202 Accepted` Response Code

A service MAY reply to a Data Modification Request with `202 Accepted`, indicating that the request has been accepted but has not yet completed. In this case, the response body MUST contain a `Location` header in addition to a `Retry-After` header, and the response body MUST be empty.

Once the request has successfully completed, the service MUST return `303 See Other` with a `Location` header specifying the final URL to retrieve the outcome of the operation. The response body and headers from this final URL MUST be formatted as would the completion of the initial Data Modification Request.

9.1.4. `204 No Content` Response Code

A service may reply to a Data Modification Request with `204 No Content`. In this case, the response body MUST be empty.

9.1.5. `3xx Redirect` Response Code

As per RFC2616 (https://www.w3.org/Protocols/rfc2616/rfc2616.html), an OData service MAY respond to any Data Modification request with a response code of `3xx Redirection`. In this case, the response SHOULD include a `Location` header with the URL from which the result can be obtained.

The service MUST ensure that no observable change has occurred to the state of the service as a result of any request that returns a `3xx`.

9.2. `4xx Client Error` Responses

Services SHOULD return `4xx Client Error` in response to client errors such as malformed requests, in addition to requests for non-existent resources such as entity collections, entities, or properties.

The service MUST ensure that no observable change has occurred to the state of the service as a result of any request that returns an error status code.

In the case that a response body is defined for the error code, the body of the error is as defined for the appropriate format.

9.2.1. `404 Not Found` Response Code

If the entity or collection specified by the request URL does not exist, the service SHOULD respond with `404 Not Found` and an empty response body.

9.3 InStream Errors

In the case that the service encounters an error after sending a success status to the client, the service MUST generate an in-stream error which SHOULD leave the response malformed. Clients MUST assume that any malformed responses are invalid and results SHOULD be discarded.

This specification does not prescribe a particular format for such instream errors.

10. OData Service Requests

An OData Service MAY support the following types of requests.

10.1. Metadata Requests

An OData service is a self-describing service that exposes metadata defining the entity sets, relationships, entity types, and operations.

10.1.1. Service Document Request

To request a `Service Document` describing the set of entity collections (i.e., entity sets) that can be queried from a service, the client issues a GET request to the root URL of the service (the *Service Root*).

The format of the Service Document is dependent upon the format selected. For example, in Atom the Service Document is an AtomPub Service Document (as specified in RFC5023 (https://tools.ietf.org/html/rfc5023)).

10.1.2. Metadata Document Request

An OData *Metadata Document* is a representation of the data model that describes the data and operations exposed by an OData service.

OData:CSDL (/documentation/odata-version-3-0/common-schema-definition-language-csdl) describes an XML representation for OData Metadata Documents and provides an XSD to validate their contents. The media type of the XML representation of an OData Metadata Document is 'application/xml'.

OData services MUST expose a Metadata Document which defines all data exposed by the service. The *Metadata Document URL* SHOULD be the root URL of the service with "/$metadata" appended. To retrieve this document a client issues a GET request to the Metadata Document URL.

If a request for metadata does not specify a format preference (via `Accept` header or $format) then the XML representation MUST be returned.

10.2. Requesting Data

OData services support requesting data through the use of HTTP GET requests.

The path of the URL specifies the target of the request (for example; the collection of entities, entity, navigation property, scalar property, or operation). Additional query operators, such as filter, sort, page, and projection operations are specified through query options.

The format of the returned data is dependent upon the request and the format specified by the client, either in the `Accept` header or using the $format query option.

This section describes the types of data requests defined by OData. For complete details on the syntax for building requests, see OData:URL (/documentation/odata-version-3-0/url-conventions).

10.2.1. Requesting Individual Entities

To retrieve an individual entity, a client makes a GET request to an *entity request URL*.

The entity request URL MAY be returned in a response payload containing that instance (for example, as a self-link in an Atom Payload (/documentation/odata-version-3-0/atom-format)).

Services MAY support conventions for constructing an entity request URL using the entity's Key Value(s), as described in OData:URL (/documentation/odata-version-3-0/url-conventions).

10.2.2. Requesting Individual Properties

A service SHOULD support retrieving an individual property value.

To retrieve an individual property, a client issues a GET request to the property URL. The property URL is the entity request URL with " `/` " and the property name appended.

For complex typed properties, the path MAY be further extended with the name of the individual property of the complex type.

See OData:URL (/documentation/odata-version-3-0/url-conventions) for details.

For example:

```
https://services.odata.org/OData/OData.svc/Products(1)/Name
```

10.2.2.1. Requesting a Property's Raw Value using `$value`

A service SHOULD support retrieving the raw value of a primitive type property. To retrieve this value, a client sends a GET request to the property value URL. See the OData:URL (/documentation/odata-version-3-0/url-conventions) document for details.

For example:

```
https://services.odata.org/OData/OData.svc/Products(1)/Name/$value
```

The raw value of an Edm.Binary property MUST be serialized as an unencoded byte stream.

The raw value of other properties SHOULD be represented using the `text/plain` media type. See OData:ABNF (/documentation/odata-version-3-0/abnf) for details.

A `$value` request for a property that is `NULL` SHOULD result in a `404 Not Found`. response.

10.2.3. Querying Collections

OData services support querying collections of entities.

The target collection is specified through a URL, and query operations such as filter, sort, paging, and projection are specified as *System Query Options* provided as query options. The names of all System Query Options are prefixed with a "$" character.

An OData service MAY support some or all of the System Query Options defined. If a data service does not support a System Query Option, it MUST fail any request that contains the unsupported option.

10.2.3.1. The `$filter` System Query Option

The set of entities returned MAY be restricted through the use of the `$filter` System Query Option.

For example:

```
https://services.odata.org/OData/OData.svc/Products?$filter=Price lt 10.00
```

Returns all Products whose Price is less than $10.00.

The value of the `$filter` option is a boolean expression as defined in OData:ABNF (/documentation/odata-version-3-0/abnf).

10.2.3.1.1. Built-in Filter Operations

OData supports a set of built-in filter operations, as described in this section. For a full description of the syntax used when building requests, see OData:URL (/documentation/odata-version-3-0/url-conventions).

Logical Operators

Operator	Description	Example
eq	Equal	/Suppliers?$filter=Address/City eq 'Redmond'
ne	Not equal	/Suppliers?$filter=Address/City ne 'London'
gt	Greater than	/Products?$filter=Price gt 20
ge	Greater than or equal	/Products?$filter=Price ge 10
lt	Less than	/Products?$filter=Price lt 20
le	Less than or equal	/Products?$filter=Price le 100
and	Logical and	/Products?$filter=Price le 200 and Price gt 3.5
or	Logical or	/Products?$filter=Price le 3.5 or Price gt 200
not	Logical negation	/Products?$filter=not endswith(Description,'milk')

Arithmetic Operators

Operator	Description	Example
add	Addition	/Products?$filter=Price add 5 gt 10
sub	Subtraction	/Products?$filter=Price sub 5 gt 10
mul	Multiplication	/Products?$filter=Price mul 2 gt 2000
div	Division	/Products?$filter=Price div 2 gt 4
mod	Modulo	/Products?$filter=Price mod 2 eq 0

Grouping Operators

Operator	Description	Example
( )	Precedence grouping	/Products?$filter=(Price sub 5) gt 10

10.2.3.1.2. Built-in Query Functions

OData supports a set of built-in functions that can be used within `$filter` operations. The following table lists the available functions. For a full description of the syntax used when building requests, see OData:URL (/documentation/odata-version-3-0/url-conventions).

OData does note define an ISNULL or COALESCE operator. Instead, there is a `null` literal which can be used in comparisons.

String Functions

Function	Example

Function	Example
bool substringof(string searchString, string searchInString)	substringof('Alfreds',CompanyName)
bool endswith(string string, string suffixString)	endswith(CompanyName,'Futterkiste')
bool startswith(string string, string prefixString)	startswith(CompanyName,'Alfr')
int length(string string)	length(CompanyName) eq 19
int indexof(string searchInString, string searchString)	indexof(CompanyName,'lfreds') eq 1
string replace(string searchInString, string searchString, string replaceString)	replace(CompanyName,' ', '') eq 'AlfredsFutterkiste'
string substring(string string, int pos)	substring(CompanyName,1) eq 'lfreds Futterkiste'
string substring(string string, int pos, int length)	substring(CompanyName,1, 2) eq 'lf'
string tolower(string string)	tolower(CompanyName) eq 'alfreds futterkiste'
string toupper(string string)	toupper(CompanyName) eq 'ALFREDS FUTTERKISTE'
string trim(string string)	trim(CompanyName) eq 'Alfreds Futterkiste'
string concat(string string1, string string2)	concat(concat(City,', '), Country) eq 'Berlin, Germany'

Date Functions

Function	Example
int day(DateTime datetimeValue)	day(BirthDate) eq 8
int hour(DateTime datetimeValue)	hour(BirthDate) eq 1
int minute(DateTime datetimeValue)	minute(BirthDate) eq 0
int month(DateTime datetimeValue)	month(BirthDate) eq 12
int second(DateTime datetimeValue)	second(BirthDate) eq 0
int year(DateTime datetimeValue)	year(BirthDate) eq 1948

Math Functions

Function	Example
double round(double doubleValue)	round(Freight) eq 32
decimal round(decimal decimalValue)	round(Freight) eq 32
double floor(double doubleValue)	floor(Freight) eq 32
decimal floor(decimal datetimeValue)	floor(Freight) eq 32
double ceiling(double doubleValue)	ceiling(Freight) eq 33
decimal ceiling(decimal datetimeValue)	ceiling(Freight) eq 33

Type Functions

Function	Example
bool IsOf(type value)	isof('NorthwindModel.Order')
bool IsOf(expression value, type targetType)	isof(ShipCountry,'Edm.String')

10.2.3.1.3 The `$expand` System Query Option

The presence of the `$expand` system query option indicates that entities related to the entity, or collection of entities, identified by the resource path section of the URL MUST be represented inline.

The value of the `$expand` query option MUST be a comma separated list of navigation property paths.

The service MUST include any actions or functions that are bound to the associated entities that are introduced via `$expand`, unless a `$select` System Query Option is also included in the request and that `$select` requests that the actions/functions be omitted.

For a full description of the syntax used when building requests, see OData:URL (/documentation/odata-version-3-0/url-conventions).

Examples

```
http://host/service.svc/Customers?$expand=Orders
```

For each customer entity within the Customers entity set, the value of all associated Orders MUST be represented inline.

```
http://host/service.svc/Orders?$expand=OrderLines/Product,Customer
```

For each Order within the Orders entity set, the following MUST be represented inline:

- The Order lines associated to the Orders identified by the resource path section of the URL and the products associated to each Order line.
- The customer associated with each Order returned.http://host/service.svc/Customers?$expand=SampleModel.VipCustomer/InHouseStaff

For each Customer entity in the Customers entity set, the value of all associated InHouseStaff MUST be represented inline if the entity is of type VipCustomer or a subtype of that. For entities that are not of type VipCustomer, or any of its subtypes, that entity SHOULD be returned with no inline representation for the expanded navigation property.

10.2.3.2 The `$select` System Query Option

The `$select` system query option requests that the service return only the properties, open properties, related properties, actions and functions explicitly requested by the client. The service MUST return the specified content, and MAY choose to return additional information.

The value of the $select query option is a comma separated list of property paths, qualified action names, qualified function names, or the star operator (*), or the star operator prefixed with the name of the entity container in order to specify all operations within the container.

For example, the following request returns just the Rating and ReleaseDate for the matching Products:

```
https://services.odata.org/OData/OData.svc/Products?$select=Rating,ReleaseDate
```

It is also possible to request all properties, using a star request:

```
https://services.odata.org/OData/OData.svc/Products?$select=*
```

A star request SHOULD NOT introduce actions or functions not otherwise requested.

Properties of related entities MAY be specified by providing a property path in the select list. In order to select properties from related entities, the appropriate navigation property MUST be specified through the `$expand` query option:

```
https://services.odata.org/OData/OData.svc/Products?$select=*,Category/Name&$expand
=Category
```

It is also possible to request all actions and functions available for each returned entity:

```
https://services.odata.org/OData/OData.svc/Products?$select=DemoService.*
```

For AtomPub formatted responses, the value of a `$select` clause applies only to the properties returned within the `m:properties` element. For example, if a property of an entity type is mapped to an Atom element, then that property MUST always be included in the response according to its customizable feed mapping.

10.2.3.3 The `$orderby` System Query Option

The `$orderby` System Query option specifies the order in which entities are returned from the service.

The value of the `$orderby` System Query option contains a comma separated list of property navigation paths to sort by, where each property navigation path terminates on a primitive property.

A type cast using the qualified entity type name is required to order by a property defined on a derived type.

The property name 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.

For example:

```
https://services.odata.org/OData/OData.svc/Products?$orderby=ReleaseDate asc, Ratin
g desc
```

10.2.3.4. The `$top` System Query Option

The `$top` System Query Option specifies that only the first n records SHOULD be returned, where n is a non-negative integer value specified by the `$top` query option.

For example:

```
https://services.odata.org/OData/OData.svc/Products?$top=5
```

Would return only the first five Products in the Products entity set.

If no unique ordering is imposed through an `$orderby` query option, the service MUST impose a stable ordering across requests that include `$top` .

10.2.3.5. The `$skip` System Query Option

The `$skip` System Query Option specifies that the result MUST NOT include the first n entities, where n is a non-negative integer value specified by the `$skip` query option.

For example:

```
https://services.odata.org/OData/OData.svc/Products?$skip=5
```

Would return Products starting with the 6th Product in the Products entity set.

Where [`$top`](thetopsystemqueryoption] and `$skip` are used together, the `$skip` MUST be applied before the `$top` , regardless of the order in which they appear in the request.

For example:

```
https://services.odata.org/OData/OData.svc/Products?$top=5&$skip=2
```

Would return the third through seventh Products in the Products entity set.

If no unique ordering is imposed through an `$orderby` query option, the service MUST impose a stable ordering across requests that include `$skip` .

10.2.3.6. The `$inlinecount` System Query Option

The `$inlinecount` System Query Option with a value of `allpages` specifies that the total count of entities matching the request MUST be returned along with the result.

For example:

```
https://services.odata.org/OData/OData.svc/Products?$inlinecount=allpages
```

Would return, along with the results, the total number of products in the set.

An `$inlinecount` query option with a value of `none` (or not specified) hints that the service SHOULD NOT return a count.

The service MUST return an HTTP Status code of `404 Bad Request` if a value other than `allpages` or `none` is specified.

`$inlinecount` ignores any `$top` , `$skip` , or `$expand` query options, and returns the total count of results across all pages including only those results matching any specified `$filter` .

How the count is encoded in the response body is dependent upon the selected format.

10.2.3.7. The `$format` System Query Option

A request with a `$format` system query option specifies that the response SHOULD use the media type specified by the query option.

If the `$format` query option is present in a request, it SHOULD take precedence over the value(s) specified in the accept request header.

- If the value of the query option is “atom”, then the media type used in the response MUST be “application/atom+xml”, or “application/atomsvc+xml” for the service document.
- If the value of the query option is “json”, then the media type used in the response MUST be “application/json”.
- If the value of the query option is “xml”, then the media type used in the response MUST be “application/xml”.

For example:

```
http://host/service.svc/Orders?$format=json
```

Is equivalent to a request with the “accept” header set to “application/json”; it requests the set of Order entities represented using the JSON media type, as specified in OData:JSON (../json-verbose-format).

The `$format` query option MAY be used in conjunction with `$value` to specify which raw format is returned.

```
http://host/service.svc/Orders(1)/ShipCountry/$value/?$format=json
```

The raw value of the ShipCountry property of the matching Order using the JSON media type.

10.2.4. Requesting `$links` between Entities

To request the links (URLs) of related entities according to a particular relationship, the client issues a GET request.

The path of the request is `/$links/` appended to the path of a the source entity's request URL, followed by the name of the navigation property representing the relationship.

On success, the response body MUST contain the URL for each related entity, formatted as a either a single link, or a collection of links, depending on the cardinality of the relationship.

If the navigation property does not exist on the entity indicated by the request URL, the service SHOULD return `404 Not Found`.

For example:

```
https://services.odata.org/OData/OData.svc/Products(0)/$links/Orders
```

Returns the URLs of each Order related to the Product with `ID=0`.

10.2.5. Requesting the `$count` of an Entity Collection

To request only the count of an entity collection, the client issues a GET request with `/$count` appended to the path of the request URL.

On success, the response body MUST contain the count of entities matching the request, formatted as a simple scalar integer value.

For example:

```
https://services.odata.org/OData/OData.svc/Products/$count
```

Returns the count of Products in the Products entity set.

10.3. Data Modification

An OData service MAY support Create, Update, and Delete operations for some or all of the entities that it exposes. Additionally, services MAY support one or more Actions which may affect the state of the system.

A successfully completed Data Modification request must not violate the integrity of the data.

A client may request whether content be returned from a Create, Update, or Delete request, or the invocation of an Action, by specifying the `Prefer` Header.

10.3.1. Common Data Modification Semantics

Data modification requests share the following semantics.

10.3.1.1 Use of ETags for Avoiding Update Conflicts

A client MAY include an ETag value in a the `if-match` or the `if-none-match` request header of a Data Modification or Action request. If specified, the operation MUST only be invoked if the `if-match` or `if-none-match` condition is satisfied.

The ETag value specified in the `if-match` or `if-none-match` request header may be obtained from an `ETag` header of a request for an individual entity, or may be included for an individual entry in a format-specific manner.

10.3.1.2. Differential Update

Some update requests support two types of update: replace and merge. The client chooses which to execute by which HTTP verb it sends in the request.

A PUT request indicates a replacement update. The service MUST replace all property values with those specified in the request body. Missing properties MUST be set to their default values. Missing dynamic properties MUST be removed or set to NULL.

A PATCH or MERGE indicates a differential update. The service MUST replace exactly those property values that are specified in the request body. Missing properties, including dynamic properties, MUST NOT be altered.

The semantics of PATCH are defined in [RFC 5789][]. The service MUST be compliant with that definition.

The HTTP MERGE verb is defined by this document. The remainder of this section defines the semantics for the MERGE verb. All the semantics for HTTP PUT apply to HTTP MERGE. The only difference is client intent.

Because PATCH is a standard verb and MERGE is not, a client SHOULD prefer PATCH.

The semantics of a MERGE request on a data service entity is to merge the content in the request payload with the entity's current state. The merging is done by comparing each component of the request body to the entity as it exists in the service.

If a component in the request body is not defined on the entity that is to be updated the request MAY be considered malformed.

If a component in the request body does match a component on the entity that is to be updated, the value of the component in the request body MUST replace the matching component of the entity to be updated and the matching process continues with the children of the component from the request body.

10.3.2. Create an Entity

To create an entity in a collection, send a POST request to that collection's URL. The POST body MUST contain a single valid entity representation.

To create an *open entity* (an instance of an open type), additional property values beyond those specified in the metadata MAY be sent in the request body. The service MUST treat these as dynamic properties and add them to the created instance.

If the entity being created is not an open entity, additional property values beyond those specified in the metadata SHOULD NOT be sent in the request body. The service SHOULD ignore any such values supplied.

Upon successful completion, the response MUST contain a `Location` header that contains the edit URL of the created entity.

Upon successful completion the service MUST respond with either `201 Created`, or '204 No Content' if the request included a `Prefer` header with a value of "return-no-content'.

10.3.2.1. Link to Related Entities When Creating an Entity

A service SHOULD support linking new entities to existing entities upon creation.

A request to create an entity MAY specify that the entity should be linked to existing entities. To bind the new entity to existing entities, include the required relationship link in the appropriate navigation property in the request body.

The representation for binding information is format specific.

On success, the service MUST create the requested entity and relate it to the requested existing entities.

On failure, the service MUST NOT create the new entity. In particular, the service MUST never create an entity in a partially-valid state (with the navigation property unset).

10.3.2.2. Create Related Entities When Creating an Entity

A service that supports creating entities SHOULD support creating related entities as part of the same request.

A request to create an entity MAY specify related entities that should also be created. The related entities MUST be represented using the appropriate inline representation of the navigation property.

On success, the service MUST create each entity and relate them.

On failure, the service MUST NOT create any of the entities.

10.3.3. Update an Entity

To update an existing entity, send a PUT, PATCH, or MERGE request to that entity's edit URL. The request body MUST contain a single valid entity representation.

A service SHOULD support Differential Update for entities.

If the request contains a value for a key property, the service MUST ignore that value when applying the update.

If the entity being updated is open, then additional values for properties beyond those specified in the metadata MAY be sent in the request body. The service MUST treat these as dynamic properties.

If the entity being updated is not open, then additional values for properties beyond those specified in the metadata SHOULD NOT be sent in the request body. The service SHOULD ignore any such values supplied.

On success, the response MUST be a valid success response.

10.3.4. Delete an Entity

To delete an existing entity, send a DELETE request to that entity's edit URL. The request body SHOULD be empty.

On success, the response MUST be `204 No Content`.

10.3.5. Modifying Relationships Between Entities

Relationships between entities are represented by navigation properties as described in Data Model. URL conventions for navigation properties are described in OData:URL (/documentation/odata-version-3-0/url-conventions).

10.3.5.1. Create a New Link Between Two Existing Entities in a One to Many Navigation Property

To relate an existing entity to another entity, send a POST request to the URL for the appropriate navigation property's links collection. The request body MUST contain a URL that identifies the entity to be added.

The body MUST be formatted as a single link. See the appropriate format document for details.

On success, the response MUST be `204 No Content` and contain an empty body.

10.3.5.2. Remove a Relationship Between Two Entities

To remove a relationship to a related entity, send a `DELETE` request to a URL that represents the link to the related entity.

The `DELETE` request MUST follow the requirements for integrity constraints above.

On success, the response MUST be `204 No Content` and contain an empty body.

10.3.5.3. Change the Relation in a One to One Navigation Property

If a navigation property is nullable, then a change MAY be performed by first removing the existing relationship and then adding the new one. Use the approach described for adding and removing links.

Alternatively, a relationship MAY be updated as part of an update to the source entity by including the required binding information for the new target entity. This binding information MUST be formatted as for a deferred navigation property in a response.

10.3.6 Managing Media Entities

A *media entity* is an entity that represents an out-of-band stream, such as a photograph.

A media entity MUST have a `source url` that can be used to read the media stream, and MAY have an `edit-media` URL that can be used to write to the media stream.

Because a media entity has both a media stream and standard entity properties special handling is required.

10.3.6.1. Creating a Media Entity

To create a media entity, send a POST request to the media entity's entity set. The request body MUST contain the media value (for example, the photograph) in the appropriate media type.

On successful creation of the media, the service MUST respond with `201 Created` and a response body containing the newly created media entity.

10.3.6.2. Editing a Media Entity Stream

To change the data for a media entity stream, the client sends a PUT request to the edit URL of the media entity.

If the entity includes an ETag value, the client SHOULD include an `If-Match` header with the ETag value.

The request MUST contain a `Content-Type` header, set to the correct value.

The body of the request MUST be the binary data that will be the new value for the stream.

10.3.6.3. Deleting a Media Entity

To delete a media entity, send a DELETE request to the entity's edit link as described in Delete An Entity.

Deleting a media entity also deletes the media associated with the entity.

10.3.7. Managing Named Stream Properties

An entity may have one or `named stream properties`. Named stream properties are properties of type Edm.Stream.

The values for named stream properties do not appear in the entity payload. Instead, the values are read or written through URLs.

Named streams are not deletable or directly creatable by the client. The service owns their lifetime. The client can request to set the stream data to empty (0 bytes).

10.3.7.1. Editing Named Stream Values

To change the data for a named stream, the client sends a PUT request to the edit URL.

If the stream metadata includes an ETag value, the client SHOULD include an `If-Match` with the ETag value.

The request MUST contain a `Content-Type` header, set to the correct value.

The body of the request MUST be the binary data that will be the new value for the stream.

10.3.8. Managing Values and Properties Directly

Values and properties can be explicitly addressed with URLs. This allows them to be individually modified. See OData:URL (/documentation/odata-version-3-0/url-conventions) for details on addressing.

10.3.8.1. Update a Primitive Property

To update a value, the client MAY send a PUT, MERGE, or PATCH request to an edit URL for a primitive property. The message body MUST contain the new value, formatted as a single property according to the specified format.

Primitive properties do not support differential update. Regardless of which verb is used the service MUST replace the entire value with the value supplied in the request body.

The same rules apply whether this is a regular property or a dynamic property.

On success, the response MUST be a valid update response.

10.3.8.2. Null a Value

There are two ways to set a primitive value to NULL. The client MAY Update a Primitive Property, specifying a NULL value. Alternatively, the client MAY send a DELETE request with an empty message body to the property URL.

The service SHOULD consider a DELETE request to a non-nullable value to be malformed.

The same rules apply whether the target is the value of a regular property or the value of a dynamic property. A missing dynamic property is defined to be the same as a dynamic property with value NULL. All dynamic properties are nullable.

On success, the service MUST respond with `204 No Content` and an empty body.

10.3.8.3. Update a Complex Type

To update a complex type, send a PUT, PATCH, or MERGE request to that property's URL. The request body MUST contain a single valid representation for that type.

A service MUST support Differential Update for complex types.

On success, the response MUST be a valid update response.

10.3.8.4. Update a Collection Property

To update a value, send a PUT request to the collection property's URL. The message body MUST contain the desired new value, formatted as a collection property according to the specified format.

The service MUST replace the entire value with the value supplied in the request body.

On success, the response MUST be a valid update response.

10.4. Operations

Services MAY support custom operations. Operations (actions, functions and legacy service operations) are represented as function import elements in OData:CSDL (/documentation/odata-version-3-0/common-schema-definition-language-csdl).

10.4.1 Common rules for all operations

All operations MUST follow the rules outlined in OData:CSDL (/documentation/odata-version-3-0/common-schema-definition-language-csdl).

10.4.1.1. Entity Set Path Expression

For functions or actions that return an entity or collection of entities, the entity set associated with the returned entity or collection of entities MAY depend upon the entity set of one of the parameter values used to invoke the operation.

When such a dependency exists an *entity set path expression* is used. An entity set path expression MUST begin with the name of a parameter to the operation, and optionally includes a series of navigation properties (and occasional type casts) as a succinct way to describe the series of entity set transitions.

The actual entity set transitions can be deduced by finding the association set backing each navigation property, and moving from the current entity set which will be found on one end to the entity set found on the other End.

The entity set of the results of an operation invocation with an entity set path expression can only be established once the entity set of the parameter that begins the entity set path expression is known.

For example this entity set path expression: "p1/Orders/Customer" can only be evaluated once the entity set of the p1 parameter value is known.

10.4.1.2. Common Rules for Binding Operations

Actions and functions MAY be bound to an entity or a collection of entities. The first parameter of a bound operation is the *binding parameter*.

Any URL that can identify a binding parameter of the correct type MAY be used as the foundation of a URL to invoke an operation that supports binding using the resource identified by that URL as the *binding parameter value*.

For example, the function

```
<FunctionImport Name="MostRecentOrder" ReturnType="SampleModel.Order" EntitySet="Or
ders" IsBindable="true" IsSideEffecting="false" m:IsAlwaysBindable="true">
    <Parameter Name="customer" Type="SampleModel.Customer" Mode="In" />
</FunctionImport>`
```

can be bound to any url that identifies a `SampleModel.Customer` , two examples might be:

```
GET http://server/Customers(6)/MostRecentOrder()
```

Which invokes the `MostRecentOrder` function with the 'customer' or binding parameter value being the entity identified by http://server/Customers(6)/.

```
GET http://server/Contacts(23123)/Company/MostRecentOrder()
```

Which again invokes the `MostRecentOrder` function, this time with the 'customer' or binding parameter value being the entity identified by http://server/Contacts(23123)/Company/.

10.4.1. Actions

Actions are operations exposed by an OData service that MAY have side effects when invoked. Actions MAY return data and are not composable.

10.4.1.1. Declaring Actions in Metadata

Actions SHOULD be declared in `$metadata` using a `FunctionImport` element that indicates the signature (name, return type and parameters) of the action.

OData:CSDL (/documentation/odata-version-3-0/common-schema-definition-language-csdl) specifies how functions are syntactically defined in CSDL.

For example this `FunctionImport` represents an action that creates an order for a customer using the specified quantity and discount code. This action can be bound to any resource path that represents a `Customer` entity:

```
<FunctionImport Name="CreateOrder" IsBindable="true" IsSideEffecting="true"
                metadata:IsAlwaysBindable="true">
    <Parameter Name="customer" Type="SampleModel.Customer" Mode="In">
    <Parameter Name="quantity" Type="Edm.Int32" Mode="In">
    <Parameter Name="discountCode" Type="Edm.String" Mode="In">
</FunctionImport>
```

10.4.1.2. Advertising Currently Available Actions

OData:Atom (/documentation/odata-version-3-0/atom-format) and OData:JSON (/documentation/odata-version-3-0/json-verbose-format) formats require all actions that are available for the current entity or current collection of entities to be advertised inside any representation of the entity or collection entities returned from the service.

A service SHOULD advertise only those actions that are available for a given entity or collection of entities. The service MAY advertise all actions for a given entity or collection of entities. The service MAY fail later if the client attempts to invoke the action and it is found to be not available.

The following information MUST be included, as defined within the appropriate format, when an action is advertised:

- A 'Target Url' that MUST identify the resource that accepts requests to invoke the action.
- A 'Metadata Url' that MUST identify the `FunctionImport` that declares the action. This URL can be either relative or absolute, but when relative it MUST be assumed to be relative to the `$metadata` URL of the current service.
- A 'Title' that SHOULD contain a human readable description of the action.

Example: Given a GET request to http://server/Customers('ALFKI')

The service might respond with a Customer entity that advertises a binding of the `SampleEntities.CreateOrder` action to itself:

```
{"d":
 {
   "__metadata": {
       ...,
       "actions" : {
           "SampleEntities.CreateOrder" : [{
               "title" : "Create Order",
               "target" : "Customers('ALFKI')/SampleEntities.CreateOrder"
           }]
       }
   },
   "CustomerID": "ALFKI",
   "CompanyName": "Alfreds Futterkiste",
   ...
 }
}
```

When the resource retrieved represents a collection, the 'Target Url' of any actions advertised MUST encode every System Query Option used to retrieve the collection. In practice this means that any of these System Query Options should be encoded: $filter, $expand, $orderby, $skip and $top.

An efficient format that assumes client knowledge of metadata SHOULD NOT advertise Actions that are available on all instances and whose target url can be established via metadata.

10.4.1.3. Invoking an Action

To invoke an action a client MUST make a POST request to the 'Target Url' of the action.

If the action supports binding the binding parameter value MUST be encoded in the 'Target Url'. It is not possible to specify an entity or a collection of entities as a parameter value in the request body.

If the invoke request contains any non-binding parameter values, the `Content-Type` of the request MUST be `'application/json'`, and the parameter values MUST be encoded in a single JSON object in the request body.

Each non-binding parameter value specified MUST be encoded as a separate 'name/value' pair in a single JSON object that comprises the body of the request. The name is the name of the parameter. The value is the parameter value which is an instance of the type specified by the parameter in JSON format. Any parameter values not specified in the JSON object MUST be assumed to be null.

If the action returns results the client SHOULD use content type negotiation to request the results in the desired format, otherwise the default content type will be used.

If a client only wants an action to be processed when the binding parameter value, an entity or collection of entities, is unmodified, the client SHOULD include the `If-Match` header with the latest known ETag value for the entity or collection of entities.

On success, the response SHOULD be 200 for actions with a return type or 204 for action without a return type. The client can request whether any results from the action be returned using the `Prefer` header.

Example: The following request invokes the `SampleEntities.CreateOrder` action using `/Customers('ALFKI')` as the customer (or binding parameter). The values `2` for the `quantity` parameter and `BLACKFRIDAY` for the `discountcode` parameter are passed in the body of the request:

```
POST http://server/Customers('ALFKI')/SampleEntities.CreateOrder
{
   "quantity": 2,
   "discountCode": "BLACKFRIDAY"
}
```

10.4.1.4. Action Overload Resolution

Actions support overloads, meaning a service MAY expose multiple actions with the same name that take a different set of parameters.

The combination of the action name, the binding parameter type and the unordered list of non-binding parameter names MUST be sufficient to uniquely identify a specific action overload.

10.4.2. Functions

Functions are operations exposed by an OData service that MUST return data and MUST have no observable side effects.

10.4.2.1. Declaring Functions in Metadata

Functions SHOULD be declared in `$metadata`. Function declarations indicate the signature (Name, ReturnType and Parameters) and semantics (composability, bindability and result entity set) of the Function.

OData:CSDL (/documentation/odata-version-3-0/common-schema-definition-language-csdl) specifies how functions are syntactically defined in CSDL.

For Example: The following `FunctionImport` describes a Function called MostRecent that returns the most recent Order within a collection of Orders:

```
<FunctionImport Name="MostRecent" EntitySet="Orders" ReturnType="SampleModel.Order"
    IsBindable="true" IsSideEffecting="false" metadata:IsAlwaysBindable="true">
    <Parameter Name="orders" Type="Collection(SampleModel.Order)" Mode="In"/>
</FunctionImport>
```

10.4.2.2. Advertising Currently Available Functions within a Payload

Functions MAY be bound to an entity or collection of entities. Services MAY choose whether to advertise functions within the entities or collections of entities returned from the Server.

Example: Given a GET request to `http://server//Orders`, the service might respond with a collection of Orders that advertises the `SampleEntities.MostRecent` Function bound to the collection:

```
{
    "__metadata": {
        "functions": "SampleEntities.MostRecent" : [
            {
               "title" : "Most Recent Order",
               "target" : "Orders/SampleEntities.MostRecent"
            }
        ]
    },
    "d": [
         {
            "__metadata": {
                            ...
                          },
            "OrderID": 1,
            "ShippedDate": "/Date(872467200000)/",
            ...
         },
         ...
    ]
}
```

When the resource retrieved represents a collection, the 'Target Url' of any functions advertised MUST encode every System Query Option used to retrieve the collection. In practice this means that any of these System Query Options should be encoded: $filter, $expand, $orderby, $skip and $top.

An efficient format that assumes client knowledge of metadata SHOULD NOT advertise Functions that are available on all instances and whose target url can be established via metadata.

10.4.2.3. Invoking a Function

To invoke a function directly a client MUST issue a GET request to a URL that identifies the function and that specifies any parameter values required by the function.

It is also possible to invoke a function indirectly using GET, PUT, POST, PATCH, MERGE or DELETE requests by formulating a URL that identifies a function and its parameters and then appending further path segments to create a request URL that identifies resources related to the results of the function.

Parameter values passed to functions MUST be specified either as a URL Literal (for Primitive Types) or as a JSON formatted OData object (for Complex Types or Collections of Primitive Types or Complex Types).

Functions calls MAY be present in the request URL path or the request URL query inside either the

`$filter` or `$orderby` System Query Options.

10.4.2.3.1. Inline Parameter Syntax

The simplest way to pass parameter values to a function is using inline parameter syntax.

To use Inline Parameter Syntax, whereever a function is called, parameter values MUST be specified inside the parenthesis, i.e. `()` , appended directly to the function name.

The parameter values MUST be specified as a comma separated list of Name/Value pairs in the format `Name=Value` , where `Name` is the name of the parameter to the function and `Value` is the parameter value.

For example this request:

```
GET http://server/Sales.GetEmployeesByManager(ManagerID=3)
```

Invokes a `Sales.GetEmployeesByManager` function which takes a single `ManagerID` parameter.

And this request:

```
GET http://server/Customers?$filter=Sales.GetSalesRegion(City=$it/City) eq "Wester
n"
```

Filters `Customers` to those in the `Western` sales region, calculated for each Customer in the collection by passing the Customer’s City as the `City` parameter value to the `Sales.GetSalesRegion` function.

Primitive parameters values may be provided to functions in the request URL path using inline syntax. All other parameter types MUST be provided externally.

10.4.2.3.2. Parameter Aliases

Parameters MAY be specified by substituting a `parameter alias` in place of an inline parameter to a function call. Parameters aliases are names beginning with an ampersand (`@`).

Actual parameter values MUST be specified as query options in the query part of the request URL. The query option name is the Name of the parameter alias, and the query option value is the value to be used for the specified parameter alias.

For example:

```
GET http://server/Sales.GetEmployeesByManager(ManagerID=@p1)?@p1=3
```

Parameter aliases allow the same parameter value to be used multiple times in a request and MAY be used to reference non-primitive values.

If a parameter alias referenced by a function call is not given a value in the Query part of the request URL, the value MUST be assumed to be `null` .

10.4.2.3.3. Parameter Name Syntax

The OData protocol allows parameter values for the last Function call in a Request URL Path to be specified by appending Name/Value pairs, representing each parameter Name and Value for that Function, as query strings to the Query part of the Request URL.

This enables clients to invoke Functions without parsing the advertised Target Url.

For example:

```
GET http://server/Sales.GetEmployeesByManager?ManagerID=3
```

10.4.2.4. Function overload resolution

Functions overloads are supported in OData, meaning a service MAY expose multiple Functions with the same name that take a different set of parameters.

When a function is invoked (using any of the three parameter syntaxes) the parameter names and parameter values are specified in the URL, and the parameter types can be deduced from each parameter value. The combination of the Function name and the unordered list of parameter types and names is always sufficient to identify a particular function overload.

Appendix A: Terminology

alias
A simple identifier that is typically used as a short name for a **namespace**.
alias qualified name
A qualified name that is used to refer to a **structural type**, except that the **namespace** is replaced by the alias for the **namespace**. For example, if an **entity type** called "Person" is defined in the "Model.Business" **namespace**, and that **namespace** has been given the **alias** "Self", the alias qualified name for the person **entity type** is "Self.Person".
annotation
Any custom, application-specific extension that is applied to an instance of **CSDL** through the use of custom attributes and elements that are not a part of this **CSDL** specification.
association
A named independent relationship between two **entity type** definitions. Associations in the **Entity Data Model (EDM)** are first-class concepts and are always bidirectional. Indeed, the first-class nature of associations helps distinguish the **EDM** from the relational model. Every association includes exactly two association ends.
association end
A term that specifies the **entity type** elements that are related, the roles of each of those **entity type** elements in the **association**, and the **cardinality** rules for each end of the **association**.
bound Action invocation URL
the URL that can be used to invoke a particular action bound to a particular entity or collection of entities.
bound Function invocation URL
the URL that can be used to invoke a particular function bound to a particular entity or collection of entities.
cardinality
The measure of the number of elements in a set.
collection
A grouping of one or more **EDM types** that are type compatible. A collection can be used as the return type for a **FunctionImport**.
conceptual schema definition language (CSDL)
A language that is based on XML and that can be used to define conceptual models that are based on the **EDM**.
conceptual schema definition language (CSDL) document
A document that contains a conceptual model that is described by using the **CSDL** code.
declared property
A property that is statically declared by a **Property** element as part of the definition of a **structural**

type. For example, in the context of an **entity type**, a declared property includes all properties of an **entity type** that are represented by the **Property** child elements of the **entity type** element that defines the **entity type**.

derived type

A type that is derived from the **base type**. Only **complex type** and **entity type** can define a **base type**.

dynamic property

A designation for an instance of an **open entity type** that includes additional nullable properties (of a **scalar type** or **complex type**), or navigation properties, beyond its **declared properties**. The set of additional properties, and the type of each, may vary between instances of the same **open entity type**. Such additional properties are referred to as dynamic properties and do not have a representation in a **CSDL document**.

EDM type

A categorization that includes all the following types: **EDMSimpleType**, **complex type**, **entity type**, **enumeration**, and **association**.

entity

An instance of an **entity type** that has a unique identity and an independent existence. An entity is an operational unit of consistency.

entity request URL

A URL for requesting a single entity as a top-level object (as opposed to a collection containing a single entity). An entity request URL MAY be obtained from a response payload containing that instance (for example, as a self-link in an Atom Payload (https://www.odata.org/media/30002/ODataAtomPayload)). Services MAY support conventions for constructing an entity request URL using the entity's Key Value(s), as described in OData:URL (/documentation/odata-version-3-0/url-conventions).

Entity Data Model (EDM)

A set of concepts that describes the structure of data, regardless of its stored form, as described in the Introduction (section 1).

enumeration type

A type that represents a custom enumeration that is declared by using the **EnumType** element.

facet

An element that provides information that specializes the usage of a type. For example, the precision (that is, accuracy) facet can be used to define the precision of a **DateTime property**.

identifier

A string value that is used to uniquely identify a component of the **CSDL** and is of type **SimpleIdentifier**.

in scope

A designation that is applied to an XML construct that is visible or can be referenced, assuming that all other applicable rules are satisfied. Types that are in scope include all **scalar types** and **structural type** types that are defined in **namespaces** that are in scope. **Namespaces** that are in scope include the **namespace** of the current **schema** and other **namespaces** that are referenced in the current **schema** by using the **Using** element.

namespace

A name that is defined on the **schema** and that is subsequently used to prefix **identifiers** to form the **namespace qualified name** of a **structural type**. **CSDL** enforces a maximum length of 512 characters for namespace values.

namespace qualified name

A qualified name that refers to a **structural type** by using the name of the **namespace**, followed by a period, followed by the name of the **structural type**.

nominal type

A designation that applies to the types that can be referenced. Nominal types include all primitive types and named **EDM types**. Nominal types are frequently used inline with collection in the following format: collection(nominal_type).

property

An **entity type** can have one or more properties of the specified **scalar type** or **complex type**. A property can be a **declared property** or a **dynamic property**. (In **CSDL 1.2**, **dynamic properties** are defined only for use with **open entity type** instances.)

referential constraint

A constraint on the keys contained in the **associatio**n type. The ReferentialConstraint **CSDL** construct is used for defining referential constraints.

scalar type

A designation that applies to all **EDMSimpleType** and **enumeration types**. Scalar types do not include **StructuralTypes**.

schema

A container that defines a **namespace** that describes the scope of **EDM types**. All **EDM types** are contained within some **namespace**.

schema level named element

An element that is a child element of the **schema** and contains a **Name** attribute that must have a unique value.

structural type

A type that has members that define its structure. **complex type**, **entity type**, and **association** are all StructuralTypes.

type annotation

An **annotation** of a model element that allows a term and provision of zero or more values for the properties of the term.

type term

A structured term represented as an entity or complex type.

value annotation

An **annotation** that attaches a named value to a model element.

value term

A term with a single property in EDM.

vocabulary

A schema that contains definitions of value terms and/or type terms.