OData Version 4.02. Part 2: URL Conventions

Committee Specification Draft 02

28 February 2024

This stage:

https://docs.oasis-open.org/odata/odata/v4.02/csd02/part2-url-conventions/odata-v4.02-csd02-part2-url-conventions.md (Authoritative)
https://docs.oasis-open.org/odata/odata/v4.02/csd02/part2-url-conventions/odata-v4.02-csd02-part2-url-conventions.html
https://docs.oasis-open.org/odata/odata/v4.02/csd02/part2-url-conventions/odata-v4.02-csd02-part2-url-conventions.pdf

Previous stage:

https://docs.oasis-open.org/odata/odata/v4.02/csd01/part2-url-conventions/odata-v4.02-csd01-part2-url-conventions.md (Authoritative)
https://docs.oasis-open.org/odata/odata/v4.02/csd01/part2-url-conventions/odata-v4.02-csd01-part2-url-conventions.html
https://docs.oasis-open.org/odata/odata/v4.02/csd01/part2-url-conventions/odata-v4.02-csd01-part2-url-conventions.pdf

Latest stage:

https://docs.oasis-open.org/odata/odata/v4.02/odata-v4.02-part2-url-conventions.md (Authoritative)
https://docs.oasis-open.org/odata/odata/v4.02/odata-v4.02-part2-url-conventions.html
https://docs.oasis-open.org/odata/odata/v4.02/odata-v4.02-part2-url-conventions.pdf

Technical Committee:

OASIS Open Data Protocol (OData) TC

Chairs:

Ralf Handl (ralf.handl@sap.com), SAP SE
Michael Pizzo (mikep@microsoft.com), Microsoft

Editors:

Michael Pizzo (mikep@microsoft.com), Microsoft
Ralf Handl (ralf.handl@sap.com), SAP SE
Heiko Theißen (heiko.theissen@sap.com), SAP SE

Additional artifacts:

This prose specification is one component of a Work Product that also includes:

- *OData Version 4.02 Part 1: Protocol*. https://docs.oasis-open.org/odata/odata/v4.02/csd02/part1-protocol/odata-v4.02-csd02-part1-protocol.html
- *OData Version 4.02 Part 2: URL Conventions* (this document). https://docs.oasis-open.org/odata/odata/v4.02/csd02/part2-url-conventions/odata-v4.02-csd02-part2-url-conventions.html

- ABNF components: *OData ABNF Construction Rules Version 4.02 and OData ABNF Test Cases Version 4.02*. https://docs.oasis-open.org/odata/odata/v4.02/csd02/abnf/.

Related work:

This specification replaces or supersedes:

- *OData Version 4.01. Part 2: URL Conventions*. Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. OASIS Standard. Latest stage: https://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part2-url-conventions.html.
- *OData Version 4.0. Part 2: URL Conventions*. Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. OASIS Standard. Latest stage: http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part2-url-conventions.html.

This specification is related to:

- *OData Vocabularies Version 4.0*. Edited by Michael Pizzo, Ralf Handl, and Ram Jeyaraman. Latest stage: https://docs.oasis-open.org/odata/odata-vocabularies/v4.0/odata-vocabularies-v4.0.html
- *OData Common Schema Definition Language (CSDL) JSON Representation Version 4.02*. Edited by Michael Pizzo, Ralf Handl, and Heiko Theißen. Latest stage: https://docs.oasis-open.org/odata/odata-csdl-json/v4.02/odata-csdl-json-v4.02.html
- *OData Common Schema Definition Language (CSDL) XML Representation Version 4.02*. Edited by Michael Pizzo, Ralf Handl, and Heiko Theißen. Latest stage: https://docs.oasis-open.org/odata/odata-csdl-xml/v4.02/odata-csdl-xml-v4.02.html
- *OData JSON Format Version 4.02*. Edited by Michael Pizzo, Ralf Handl, and Heiko Theißen. Latest stage: https://docs.oasis-open.org/odata/odata-json-format/v4.02/odata-json-format-v4.02.html
- *OData Data Aggregation Extension Version 4.0*. Edited by Ralf Handl, Hubert Heijkers, Gerald Krause, Michael Pizzo, Heiko Theißen, and Martin Zurmuehl. Latest stage: https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/odata-data-aggregation-ext-v4.0.html
- *OData Extension for Temporal Data Version 4.0*. Edited by Ralf Handl, Hubert Heijkers, Gerald Krause, Michael Pizzo, Heiko Theißen, and Martin Zurmuehl. Latest stage: https://docs.oasis-open.org/odata/odata-temporal-ext/v4.0/odata-temporal-ext-v4.0.html

Abstract:

The Open Data Protocol (OData) enables the creation of REST-based data services, which allow resources, identified using Uniform Resource Locators (URLs) and defined in an Entity Data Model (EDM), to be published and edited by Web clients using simple HTTP messages. This specification defines a set of recommended (but not required) rules for constructing URLs to identify the data and metadata exposed by an OData service as well as a set of reserved URL query string operators.

Status:

This document was last revised or approved by the OASIS Open Data Protocol (OData) TC on the above date. The level of approval is also listed above. Check the “Latest stage” location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://groups.oasis-open.org/communities/tc-community-home2?CommunityKey=e7cac2a9-2d18-4640-b94d-018dc7d3f0e2#technical.

TC members should send comments on this specification to the TC’s email list. Any individual may submit comments to the TC by sending email to Technical-Committee-Comments@oasis-open.org. Please use a Subject line like “Comment on OData URL Conventions”.

This specification is provided under the RF on RAND Terms Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be

essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (https://www.oasis-open.org/committees/odata/ipr.php).

Note that any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Key words:

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 **[RFC2119]** and **[RFC8174]** when, and only when, they appear in all capitals, as shown here.

Citation format:

When referencing this specification the following citation format should be used:

[OData-v4.02-Part2]

OData Version 4.02. Part 2: URL Conventions. Edited by Michael Pizzo, Ralf Handl, and Heiko Theißen. 28 February 2024. OASIS Committee Specification Draft 02. https://docs.oasis-open.org/odata/odata/v4.02/csd02/odata-v4.02-csd02-part2-url-conventions.html. Latest stage: https://docs.oasis-open.org/odata/odata/v4.02/odata-v4.02-part2-url-conventions.html.

Notices

Table of Contents

1 Introduction

The Open Data Protocol (OData) enables the creation of REST-based data services, which allow resources, identified using Uniform Resource Locators (URLs) and defined in a data model, to be published and edited by Web clients using simple HTTP messages. This specification defines a set of recommended (but not required) rules for constructing URLs to identify the data and metadata exposed by an OData service as well as a set of reserved URL query string operators, which if accepted by an OData service, MUST be implemented as required by this document.

The **[OData-JSON]** document specifies the format of the resource representations that are exchanged using OData and the **[OData-Protocol]** document describes the actions that can be performed on the URLs (optionally constructed following the conventions defined in this document) embedded in those representations.

Services are encouraged to follow the URL construction conventions defined in this specification when possible as consistency promotes an ecosystem of reusable client components and libraries.

1.1 Changes from Earlier Versions

Section	Feature / Change	Issue
Section 4.17	**`POST ~/$query`** with **`Content-Type: application/x-www-form-urlencoded`** or **`application/json`**	320, 371
Section 5.1.1.7.1	New overload for function **`matchespattern`** with flags	441
Section 5.1.3	Nested query options can only appear once per expand item	2004
Section 5.1.8	Allow alternative **`$search`** syntax	293

1.2 Glossary

1.2.1 Definitions of Terms

1.2.2 Acronyms and Abbreviations

1.2.3 Document Conventions

Keywords defined by this specification use **`this monospaced font`**.

Function signatures in this specification use the following paragraph style:

```
Type FunctionName(Type Param1, Type Param2)
```

Some sections of this specification are illustrated with non-normative examples.

Example 1: text describing an example uses this paragraph style

```
Non-normative examples use this paragraph style.
```

All examples in this document are non-normative and informative only.

All other text is normative unless otherwise labeled.

2 URL Components

A URL used by an OData service has at most three significant parts: the *service root URL*, the *resource path*, and *query options*. Additional URL constructs (such as a fragment) can be present in a URL used by an OData service; however, this specification applies no further meaning to such additional constructs.

Example 2: OData URL broken down into its component parts:

```
http://host:port/path/SampleService.svc/Categories(1)/Products?$top=2&$orderby=Name
\_____________________________________/\_____________________/ \__________________/
                  |                                |                     |
          service root URL                 resource path          query options
```

Mandated and suggested content of these three significant URL components used by an OData service are covered in sequence in the three following chapters.

2.1 URL Parsing

OData follows the URI syntax rules defined in **[RFC3986]** and in addition assigns special meaning to several of the sub-delimiters defined by **[RFC3986]**, so special care has to be taken regarding parsing and percent-decoding.

[RFC3986] defines three steps for URL processing that MUST be performed before percent-decoding:

- Split undecoded URL into components scheme, hier-part, query, and fragment
- Split undecoded hier-part into authority and path
- Split undecoded path into path segments

After applying these steps defined by RFC3986 the following steps MUST be performed:

- Split undecoded query at "**`&`**" (octet **`0x26`**) into query options, and each query option at the first "**`=`**" (octet **`0x3D`**) into query option name and query option value
- Percent-decode path segments, query option names, and query option values exactly once
- Interpret path segments, query option names, and query option values according to OData rules

Note: neither **[RFC3986]** nor this specification assign special meaning to "**`+`**" (octet **`0x2B`**). Some implementations decode "**`+`**" (octet **`0x2B`**) as space (octet **`0x20`**), others take it literally.

Clients SHOULD percent-encode space (octet **`0x20`**) as **`%20`** and "**`+`**" (octet **`0x2B`**) as **`%2B`** and avoid the ambiguous "**`+`**" (octet **`0x2B`**) in URLs.

2.2 URL Syntax

The OData syntax rules for URLs are defined in this document and the **[OData-ABNF]**. Note that the ABNF is not expressive enough to define what a correct OData URL is in every imaginable use case. This specification document defines additional rules that a correct OData URL MUST fulfill. In case of doubt on what makes an OData URL correct the rules defined in this specification document take precedence. Note also that the rules in **[OData-ABNF]** assume that URLs and URL parts have been percent-encoding normalized as described in section 6.2.2.2 of **[RFC3986]** before applying the grammar to them, i.e. all characters in the unreserved set (see rule **`unreserved`** in **[OData-ABNF]**) are plain literals and not percent-encoded. For characters outside of the unreserved set that are significant to OData the ABNF rules explicitly state whether the percent-encoded representation is treated identical to the plain literal representation. This is done to make the input strings in the ABNF test cases more readable.

For example, one of these rules is that single quotes within string literals are represented as two consecutive single quotes.

Example 3: valid OData URLs:

```
http://host/service/People('O''Neil')
```

```
http://host/service/People(%27O%27%27Neil%27)
```

```
http://host/service/People%28%27O%27%27Neil%27%29
```

```
http://host/service/Categories('Smartphone%2FTablet')
```

Example 4: invalid OData URLs:

```
http://host/service/People('O'Neil')
```

```
http://host/service/People('O%27Neil')
```

```
http://host/service/Categories('Smartphone/Tablet')
```

The first and second examples are invalid because a single quote in a string literal must be represented as two consecutive single quotes. The third example is invalid because forward slashes are interpreted as path segment separators and **`Categories('Smartphone`** is not a valid OData path segment, nor is **`Tablet')`**.

3 Service Root URL

The service root URL identifies the root of an OData service. A **`GET`** request to this URL returns the format-specific service document, see **[OData-JSON, section 5]**.

The service root URL MUST terminate in a forward slash.

The service document enables simple hypermedia-driven clients to enumerate and explore the resources published by the OData service.

4 Resource Path

The rules for resource path construction as defined in this section are optional. OData services SHOULD follow the subsequently described URL path construction rules and are indeed encouraged to do so; as such consistency promotes a rich ecosystem of reusable client components and libraries.

Services that do not follow the resource path conventions for entity container children are strongly encouraged to document their resource paths by annotating entity container children with the term **Core.ResourcePath** defined in **[OData-VocCore]**. The annotation value is the URL of the annotated resource and may be relative to **xml:base** (if present), otherwise the request URL.

Resources exposed by an OData service are addressable by corresponding resource path URL components to enable interaction of the client with that resource aspect.

To illustrate the concept, some examples for resources might be: customers, a single customer, orders related to a single customer, and so forth. Examples of addressable aspects of these resources as exposed by the data model might be: collections of entities, a single entity, properties, links, operations, and so on.

An OData service MAY respond with **301 Moved Permanently** or **307 Temporary Redirect** from the canonical URL to the actual URL.

4.1 Addressing the Model for a Service

OData services expose their entity model according to **[OData-CSDL]** at the metadata URL, formed by appending **$metadata** to the service root URL.

Example 5: Metadata document URL

```
http://host/service/$metadata
```

OData clients may request a particular format for metadata either through the **Accept** header or by using the **$format** system query option.

Example 6: Metadata document URL with specified format

```
http://host/service/$metadata?$format=json
```

4.2 Addressing the Batch Endpoint for a Service

OData services that support batch requests expose a batch URL formed by appending **$batch** to the service root URL.

Example 7: batch URL

```
http://host/service/$batch
```

4.3 Addressing Entities

The basic rules for addressing a collection (of entities), a single entity within a collection, a singleton, as well as a property of an entity are covered in the **resourcePath** syntax rule in **[OData-ABNF]**.

Below is a (non-normative) snippet from **[OData-ABNF]**:

```
resourcePath = entitySetName                  [collectionNavigation]
             / singletonEntity                [singleNavigation]
             / actionImportCall
```

```
              / entityColFunctionImportCall    [ collectionNavigation ]
              / entityFunctionImportCall       [ singleNavigation ]
              / complexColFunctionImportCall   [ collectionPath ]
              / complexFunctionImportCall      [ complexPath ]
              / primitiveColFunctionImportCall [ collectionPath ]
              / primitiveFunctionImportCall    [ primitivePath ]
              / functionImportCallNoParens     [ querySegment ]
              / crossjoin                      [ querySegment ]
              / %s"$all"                       [ "/" optionallyQualifiedEntityTypeName ]
```

Since OData has a uniform composable URL syntax and associated rules there are many ways to address a collection of entities, including, but not limited to:

- Via an entity set (see rule **`entitySetName`** in **[OData-ABNF]**)

Example 8:

```
http://host/service/Products
```

- By navigating a collection-valued navigation property (see rule: **`entityColNavigationProperty`**)
- By invoking a function import that returns a collection of entities (see rule: **`entityColFunctionImportCall`**)

Example 9: function with parameters in resource path

```
http://host/service/ProductsByCategoryId(categoryId=2)
```

Example 10: function with parameters as query options

```
http://host/service/ProductsByColor(color=@color)?@color='red'
```

- By invoking an action import that returns a collection of entities (see rule: **`actionImportCall`**)

Likewise there are many ways to address a single entity.

Sometimes a single entity can be accessed directly, for example by:

- Invoking a function import that returns a single entity (see rule: **`entityFunctionImportCall`**)
- Invoking an action import that returns a single entity (see rule: **`actionImportCall`**)
- Addressing a singleton

Example 11:

```
http://host/service/BestProductEverCreated
```

Often however a single entity is accessed by composing more path segments to a **`resourcePath`** that identifies a collection of entities, for example by:

- Using an entity key to select a single entity (see rules: **`collectionNavigation`** and **`keyPredicate`**)

Example 12:

```
http://host/service/Categories(1)
```

If the declared entity type of the collection does not define a key (for example, Edm.EntityType), then a cast segment to a type defining a key MUST follow the collection, before appending the key.

Example 13:

```
http://host/service/Categories(1)/Children/myNs.Product(7)
```

- Invoking an action bound to a collection of entities that returns a single entity (see rule: **boundOperation**)
- Invoking an function bound to a collection of entities that returns a single entity (see rule: **boundOperation**)

Example 14:

```
http://host/service/Products/Model.MostExpensive()
```

These rules are recursive, so it is possible to address a single entity via another single entity, a collection via a single entity and even a collection via a collection; examples include, but are not limited to:

- By following a navigation from a single entity to another related entity (see rule: **entityNavigationProperty**)

Example 15:

```
http://host/service/Products(1)/Supplier
```

- By invoking a function bound to a single entity that returns a single entity (see rule: **boundOperation**)

Example 16:

```
http://host/service/Products(1)/Model.MostRecentOrder()
```

- By invoking an action bound to a single entity that returns a single entity (see rule: **boundOperation**)
- By following a navigation from a single entity to a related collection of entities (see rule: **entityColNavigationProperty**)

Example 17:

```
http://host/service/Categories(1)/Products
```

- By invoking a function bound to a single entity that returns a collection of entities (see rule: **boundOperation**)

Example 18:

```
http://host/service/Categories(1)/Model.TopTenProducts()
```

- By invoking an action bound to a single entity that returns a collection of entities (see rule: **boundOperation**)
- By invoking a function bound to a collection of entities that returns a collection of entities (see rule: **boundOperation**)

Example 19:

```
http://host/service/Categories(1)/Products/Model.AllOrders()
```

- By invoking an action bound to a collection of entities that returns a collection of entities (see rule: **boundOperation**)

Finally it is possible to compose path segments onto a resource path that identifies a primitive, complex instance, collection of primitives or collection of complex instances and bind an action or function that returns an entity or collections of entities.

4.3.1 Canonical URL

For OData services conformant with the addressing conventions in this section, the canonical form of an absolute URL identifying a non-contained entity is formed by adding a single path segment to the service root URL. The path segment is made up of the name of the entity set associated with the entity followed by the key predicate identifying the entity within the collection. No type-cast segment is added to the canonical URL, even if the entity is an instance of a type derived from the declared entity type of its entity set.

The canonical key predicate for single-part keys consists only of the key property value without the key property name. For multi-part keys the key properties appear in the same order they appear in the key definition in the service metadata.

Example 20: Non-canonical URL

```
http://host/service/Categories(ID=1)/Products(ID=1)
```

Example 21: Canonical URL for previous example:

```
http://host/service/Products(1)
```

4.3.2 Canonical URL for Contained Entities

For contained entities (i.e. related via a containment navigation property, see **[OData-CSDLJSON]**) the canonical URL is the canonical URL of the containing entity followed by:

- A type-cast segment if the navigation property is defined on a type derived from the entity type declared for the entity set,
- A path segment for the containment navigation property, and
- If the navigation property returns a collection, a key predicate that uniquely identifies the entity in that collection.

If the containment navigation property is defined on a complex type used within an ordered collection, the canonical URL of the containing entity is the canonical URL for the collection of complex types followed by a segment containing the zero-based ordinal of the complex typed instance within the collection.

4.3.3 URLs for Related Entities with Referential Constraints

If a navigation property leading to a related entity type has a partner navigation property that specifies a referential constraint, then those key properties of the related entity that take part in the referential constraint MAY be omitted from URLs.

Example 22: full key predicate of related entity

```
https://host/service/Orders(1)/Items(OrderID=1,ItemNo=2)
```

Example 23: shortened key predicate of related entity

```
https://host/service/Orders(1)/Items(2)
```

The two above examples are equivalent if the navigation property **`Items`** *from* **`Order`** *to* **`OrderItem`** *has a partner navigation property from* **`OrderItem`** *to* **`Order`** *with a referential constraint tying the value of the* **`OrderID`** *key property of the* **`OrderItem`** *to the value of the* **`ID`** *key property of the* **`Order`**.

The shorter form that does not specify the constrained key parts redundantly is preferred. If the value of the constrained key is redundantly specified, then it MUST match the principal key value.

4.3.4 Resolving an Entity-Id

To resolve an entity-id into a representation of the identified entity, the client issues a **`GET`** request to the **`$entity`** resource located at the URL **`$entity`** relative to the service root URL. The entity-id MUST be specified using the system query option **`$id`**. The entity-id may be expressed as an absolute IRI or relative to the request root URL.

Example 24: request the entity representation for an entity-id

```
http://host/service/$entity?$id=Products(0)
```

The semantics of **`$entity`** are covered in **[OData-Protocol, section 10]**.

4.3.5 Alternate Keys

In addition to the canonical (primary) key an entity set or entity type can specify one or more alternate keys with the **`Core.AlternateKeys`** term (see **[OData-VocCore]**).

Alternate keys can be used by the client to address entities anywhere the canonical key can be used; for example, within entity sets, collection-valued navigation properties, collection-valued composable functions, and within delta payloads.

Entities are addressed via an alternate key using the same parentheses-style convention as for the canonical key, with one difference: single-part alternate keys MUST specify the key property name to unambiguously determine the alternate key.

Example 25: the same employee identified via the alternate key SSN, the canonical (primary) key ID using the non-canonical long form with specified key property name, and the canonical short form without key property name

```
http://host/service/Employees(SSN='123-45-6789')
```

```
http://host/service/Employees(ID='A1245')
```

```
http://host/service/Employees('A1245')
```

4.3.6 Key-as-Segment Convention

Services MAY support an alternate convention for addressing entities by appending a segment containing the unprefixed and unquoted key value to the URL of the collection containing the entity. If the declared entity type of the collection does not define a key (for example, Edm.EntityType), then a cast segment to a type defining a key MUST follow the collection, before appending the key value segment. Forward-slashes in key value segments MUST be percent-encoded; single quotes within key value segments are treated as part of the key value and do not need to be doubled or percent encoded.

Example 26: valid OData URLs:

```
http://host/service/Employees/A1245
```

```
http://host/service/People/O'Neil
```

```
http://host/service/People/O%27Neil
```

```
http://host/service/Categories/Smartphone%2FTablet
```

```
http://host/service/ThingyWithDurationKey/P12DT23H59M59.999999999999S
```

```
http://host/service/ThingyWithEnumerationKey/Yellow
```

For multi-part keys, the entity MUST be addressed by multiple segments applied, one for each key value, in the order they appear in the metadata description of the entity key.

Example 27: multi-part key predicate, parentheses-style and key-as-segment

```
https://host/service/OrderItems(OrderID=1,ItemNo=2)
```

```
https://host/service/OrderItems/1/2
```

If a navigation property leading to a related entity type has a partner navigation property that specifies a referential constraint, then those key properties of the related entity that take part in the referential constraint MUST be omitted from URLs using key-as-segment convention.

Example 28: key predicate of related entity — no key segments for key properties of related entity with a referential constraint to preceding key segments

```
https://host/service/Orders/1/Items/2
```

The above example assumes that the navigation property **`Items`** *from* **`Order`** *to* **`OrderItem`** *has a partner navigation property from* **`OrderItem`** *to* **`Order`** *with a referential constraint tying the value of the* **`OrderID`** *key property of the* **`OrderItem`** *to the value of the* **`ID`** *key property of the* **`Order`**.

Because representing key values as segments could be ambiguous with other URL construction conventions, services that support key-as segment MUST implement the following precedence rules:

If a segment following an entity collection:

1. matches a defined OData segment (starting with "**`$`**"), treat it as such
2. matches a qualified bound function, bound action, or type name, treat it as such
3. matches an unqualified bound function, bound action, or type name defined in a default namespace (see **[OData-Protocol, section 4.4]**) treat it as such
4. treat as a key value

For maximum interoperability, services that support the key-as-segment convention SHOULD also support the canonical parentheses-style convention for addressing an entity within a collection, otherwise they MUST specify the URL for each returned entity in a response, as specified by the particular format.

Note: the key-as-segment convention can only be used with the canonical (primary) key and cannot be used with alternate keys as the key property names are not present in the keys and an alternative key cannot be determined.

4.4 Addressing References between Entities

OData services are based on a data model that supports relationships as first class constructs. For example, an OData service could expose a collection of Products entities each of which are related to a Category entity.

References between entities are addressable in OData just like entities themselves are (as described above) by appending a navigation property name followed by **`/$ref`** to the entity URL.

Example 29: URL addressing the references between **`Categories(1)`** *and* **`Products`**

```
http://host/service/Categories(1)/Products/$ref
```

Resource paths addressing a single related entity reference can be used in **`DELETE`** requests to unrelate two entities. In OData 4.01, this includes resource paths that identify an individual entity reference within a related collection by key. In OData 4.0, resource paths addressing a collection of references MUST be followed by the system query option **`$id`** in order to identify a single entity reference within the collection to be removed. The entity-id specified by **`$id`** may be expressed absolute or relative to the request URL. For details see **[OData-Protocol, section 4.1]**.

Example 30: three ways of unrelating **`Categories(1)`** *and* **`Products(0)`***; the second option is supported only in OData 4.01*

```
DELETE http://host/service/Categories(1)/Products/$ref?$id=../../Products(0)
```

```
DELETE http://host/service/Categories(1)/Products(0)/$ref
```

```
DELETE http://host/service/Products(0)/Category/$ref
```

4.5 Addressing Operations

The semantic rules for addressing and invoking actions and functions are defined in **[OData-Protocol, section 11.5]**.

Services MAY additionally support the use of the unqualified name of an action or function in a URL by defining one or more default namespaces through the **`Core.DefaultNamespace`** term defined in **[OData-VocCore]**. For more information on default namespaces, see **[OData-Protocol, section 4.4]**.

4.5.1 Addressing Actions

The grammar for addressing and invoking actions is defined by the following syntax grammar rules in **[OData-ABNF]**:

- The **`actionImportCall`** syntax rule defines the grammar in the **`resourcePath`** for addressing and invoking an action import directly from the service root.
- The **`boundActionCall`** syntax rule defines the grammar in the **`resourcePath`** for addressing and invoking an action that is appended to a **`resourcePath`** that identifies some resources that can be used as the binding parameter value when invoking the action.
- The **`boundOperation`** syntax rule (which encompasses the **`boundActionCall`** syntax rule), when used by the **`resourcePath`** syntax rule, illustrates how a **`boundActionCall`** can be appended to a **`resourcePath`**.

4.5.2 Addressing Functions

The grammar for addressing and invoking functions is defined by syntax rules in **[OData-ABNF]**, in particular:

- The function import call syntax rules **`complexFunctionImportCall`**, **`complexColFunctionImportCall`**, **`entityFunctionImportCall`**, **`entityColFunctionImportCall`**, **`primitiveFunctionImportCall`**, and **`primitiveColFunctionImportCall`** define the grammar in the **`resourcePath`** for addressing and providing parameters for a function import directly from the service root.
- The bound function call syntax rules **`boundComplexFunctionCall`**, **`boundComplexColFunctionCall`**, **`boundEntityFunctionCall`**, **`boundEntityColFunctionCall`**, **`boundPrimitiveFunctionCall`** and **`boundPrimitiveColFunctionCall`** define the grammar in the **`resourcePath`** for addressing and providing parameters for a function that is appended to a **`resourcePath`** that identifies some resources that can be used as the binding parameter value when invoking the function.
- The **`boundOperation`** syntax rule (which encompasses the bound function call syntax rules), when used by the **`resourcePath`** syntax rule, illustrates how a bound function call can be appended to a **`resourcePath`**.
- The **`functionExpr`** and **`boundFunctionExpr`** syntax rules as used by the **`commonExpr`** syntax rule define the grammar for invoking functions, for example to help filter and order resources identified by the **`resourcePath`** of the URL.
- The **`aliasAndValue`** syntax rule defines the grammar for providing function parameter values using Parameter Alias Syntax, see **[OData-Protocol, section 11.2.6.1.3]**.

Note: there is no literal representation for **`Edm.Stream`** values in URLs, so it is not possible to pass **`Edm.Stream`** values to parameters of function imports or to non-binding parameters of bound functions used in the resource path. Function expressions within query options can use path expressions of type **`Edm.Stream`** as values of non-binding function parameters. The aliases can contain common expressions. In the case of a bound function these MAY contain path expressions, which the service evaluates on the binding parameter value.

Example 31: An employee's leave requests for the next two weeks pending their manager's approval:

```
http://host/service/Employees(23)/self.PendingLeaveRequests(StartDate=@start,
  EndDate=@end,Approver=@approver)
  ?@start=now()
  &@end=now() add duration'P14D'
  &@approver=Manager
```

The expression **`Manager`** *is evaluated on the binding parameter value* **`Employees(23)`**.

When invoking an unbound function through a function import, expressions involving paths must start with **`$root`**:

```
http://host/service/PendingLeaveRequests(Requester=@requester,
  StartDate=@start,EndDate=@end,Approver=@approver)
  ?@requester=$root/services/Employee(23)
  &@start=now()
  &@end=now() add duration'P14D'
  &@approver=$root/services/Employee(23)/Manager
```

4.6 Addressing a Property

To address an entity property clients append a path segment containing the property name to the URL of the entity. If the property has a complex type value, properties of that value can be addressed by further property name composition.

4.7 Addressing a Raw Value

To address the raw value of a primitive property or operation result, clients append the path segment **`/$value`** to the property or operation URL.

Properties and operation results of type **`Edm.Stream`** already return the raw value of the media stream and do not support appending the **`/$value`** segment.

4.8 Addressing the Count of a Collection

To address the raw value of the number of items in a collection, clients append **`/$count`** to the resource path of the URL identifying the entity set or collection.

The **`/$count`** path suffix identifies the integer count of records in the collection and SHOULD NOT be combined with the system query options **`$top`**, **`$skip`**, **`$orderby`**, **`$expand`**, and **`$format`**. The count MUST NOT be affected by **`$top`**, **`$skip`**, **`$orderby`**, or **`$expand`**.

The count is calculated after applying any **`/$filter`** path segments, or **`$filter`** or **`$search`** system query options to the collection.

Example 32: the number of related entities

```
http://host/service/Categories(1)/Products/$count
```

Example 33: the number of entities in an entity set

```
http://host/service/Products/$count
```

Example 34: entity count in a **`$filter`** *expression. Note that the spaces around* **`gt`** *are for readability of the example only; in real URLs they must be percent-encoded as* **`%20`**.

```
http://host/service/Categories?$filter=Products/$count gt 0
```

Example 35: count of a filtered collection in a **`$filter`** *expression; returns all Categories containing more than two products whose price is greater than 5.00.*

```
http://host/service/Categories?$filter=Products/$count($filter=Price gt 5.00) gt 2
```

Example 36: entity count in an **`$orderby`** *expression*

```
http://host/service/Categories?$orderby=Products/$count
```

4.9 Addressing a Member within an Entity Collection

Collections of entities are modeled as entity sets, collection-valued navigation properties, or operation results.

For entity sets, results of operations associated with an entity set through an **`EntitySet`** or **`EntitySetPath`** declaration, or collection-valued navigation properties with a **`NavigationPropertyBinding`** or **`ContainsTarget=true`** specification, members of the collection can be addressed by convention by appending the parenthesized key to the URL specifying the collection of entities, or by using the key-as-segment convention if supported by the service.

For collection-valued navigation properties with navigation property bindings that end in a type-cast segment, a type-cast segment MUST be appended to the collection URL before appending the key segment.

Note: entity sets or collection-valued navigation properties annotated with the term **`Capabilities.IndexableByKey`** defined in **[OData-VocCap]** and a value of **`false`** do not support addressing their members by key.

4.10 Addressing a Member of an Ordered Collection

Collections can be annotated as ordered using the **`Core.Ordered`** term (see **[OData-VocCore]**). Individual items within an ordered collection of primitive or complex types can be addressed by appending a segment containing the zero-based ordinal to the URL of the collection. A negative ordinal number indexes from the end of the collection, with **`-1`** representing the last item in the collection. Services MUST NOT specify a negative index when returning the address of a member of an ordered collection.

Entity types are stably addressable using their canonical URL and SHOULD NOT be accessed or accessible using an index.

Example 37: the first address in a list of addresses for **`MainSupplier`**

```
http://host/service/MainSupplier/Addresses/0
```

4.11 Addressing Derived Types

Any resource path or path expression identifying a collection of entities or complex type instances can be appended with a *type-cast segment*, that is a path segment containing the qualified name of a type derived from the declared item type of the collection. The result will be restricted to instances of the derived type and may be empty.

Any resource path or path expression identifying a single entity or complex type instance can be appended with a type-cast segment containing the qualified name of a type derived from the declared type of the identified resource. If used in a resource path and the identified resource is not an instance of the derived type, the request will result in a **`404 Not Found`** response. If used in a path expression, the type cast will evaluate to **`null`**.

Services MAY additionally support the use of the unqualified name of a derived type in a URL by defining one or more default namespaces through the **`Core.DefaultNamespace`** term defined in **[OData-VocCore]**. For more information on default namespaces, see **[OData-Protocol, section 4.4]**.

Services MAY also support treating an instance as a type outside of the type hierarchy using the same syntax and semantics as when addressing a derived type. In this case, the set and values of properties of the addressed type may be different than the properties of the source type. The set of such possible target types outside of the type hierarchy SHOULD be called out using the **`Core.MayImplement`** annotation term, defined in **[OData-VocCore]**.

Example 38: entity set restricted to **`VipCustomer`** *instances*

```
http://host/service/Customers/Model.VipCustomer
```

Example 39: entity restricted to a **`VipCustomer`** *instance, resulting in* **`404 Not Found`** *if the customer with key* **`1`** *is not a* **`VipCustomer`**

```
http://host/service/Customers/Model.VipCustomer(1)
```

```
http://host/service/Customers(1)/Model.VipCustomer
```

Example 40: cast the complex property **`Address`** *to its derived type* **`DetailedAddress`**, *then get a property of the derived type*

```
http://host/service/Customers(1)/Address/Model.DetailedAddress/Location
```

Example 41: filter expression with type cast; will evaluate to **`null`** *for all non-*`VipCustomer` *instances and thus return only instances of* **`VipCustomer`**

```
http://host/service/Customers
  ?$filter=Model.VipCustomer/PercentageOfVipPromotionProductsOrdered gt 80
```

Example 42: expand the single related **`Customer`** *only if it is an instance of* **`Model.VipCustomer`**. *For to-many relationships only* **`Model.VipCustomer`** *instances would be inlined,*

```
http://host/service/Orders?$expand=Customer/Model.VipCustomer
```

4.12 Addressing a Subset of a Collection

Any resource path or path expression identifying a collection can be appended with a filter path segment consisting of **`/$filter`** followed by parentheses containing a parameter alias or a filter expression following the **`filter`** syntax rule in **[OData-ABNF]**. If the parentheses contain a parameter alias, a filter expression MUST be assigned to the parameter alias in the query part of the request URL. If the filter path segment appears in the resource path, the filter expression in parentheses MUST NOT use forward slashes, it must be specified with a parameter alias instead.

The collection will be restricted to instances matching the filter expression assigned to the parameter alias and may be empty.

The **`/$filter`** path segment MAY occur multiple times and it MAY be combined with the **`$filter`** system query option.

Example 43: red products that cost less than 10 — combining path segment and system query option

```
GET Products/$filter(@foo)?@foo=Price lt 10&$filter=Color eq 'red'
```

Example 44: red products that cost less than 10 — combine two path segments

```
GET Products/$filter(@p)/$filter(@c)?@p=Price lt 10&@c=Color eq 'red'
```

Example 45: categories with less than ten products older than three

```
GET Categories?$filter=Products/$filter(Age gt 3)/$count lt 10
```

Note that the **`/$filter`** path segment filters the “input” to the operation, and the **`$filter`** system query option filters the result of the operation, so the two can be used interchangeably only for **`GET`** operations. For a **`PATCH`** operation, the **`/$filter`** path segment is evaluated on the items *before* the modification and determines which items are to be modified, whereas the **`$filter`** system query option is evaluated *after* the modification and determines which of the modified items are part of the response.

The **/$filter** path segment MAY be followed by a path segment identifying a bound action or bound function applicable to the collection.

Example 46: invoke the **Special.Cluster** *action on all products older than 3*

```
POST /service/Products/$filter(@foo)/Special.Cluster?@foo=Age gt 3
```

4.13 Addressing Each Member of a Collection

To apply a **PATCH** or **DELETE** request to each member of a collection, clients append the path segment **/$each** to a resource path identifying a collection.

To apply a bound action or function to each member of a collection, clients append the path segment **/$each** followed by a path segment identifying the bound action or function. The response is a collection of instances of the result type of the bound operation. If the bound operation returns a collection, the response is a collection of collections. System query options applied to the response can use **$it** to reference an item in the outermost collection, followed by **/any**, **/all**, or **/$count**.

The resource path of the collection preceding **/$each** MAY contain type-cast segments or filter path segments to subset the collection.

4.14 Addressing the Media Stream of a Media Entity

To address the media stream represented by a media entity, clients append **/$value** to the resource path of the media entity URL. Services may redirect from this canonical URL to the source URL of the media stream.

Example 47: request the media stream for the picture with the key value **Sunset4321299432***:*

```
http://host/service/Pictures('Sunset4321299432')/$value
```

4.15 Addressing the Cross Join of Entity Sets

In addition to querying related entities through navigation properties defined in the entity model of a service, the cross join operator allows querying across unrelated entity sets.

The cross join is addressed by appending the path segment **$crossjoin** to the service root URL, followed by the parenthesized comma-separated list of joined entity sets. It returns the Cartesian product of all the specified entity sets, represented as a collection of instances of a virtual complex type. Each instance consists of one non-nullable, single-valued navigation property per joined entity set. Each such navigation property is named identical to the corresponding entity set, with a target type equal to the declared entity type of the corresponding entity set.

The **$filter** and **$orderby** query options can be specified using properties of the entities in the selected entity sets, prepended with the entity set as the navigation property name.

The **$expand** query option can be specified using the names of the selected entity sets as navigation property names. If a selected entity set is not expanded, it MUST be represented using the read URL of the related entity as a navigation link in the complex type instance.

The **$count**, **$skip**, and **$top** query options can also be used with no special semantics.

Example 48: if **Sales** *had a structural property* **ProductID** *instead of a navigation property* **Product***, a "cross join" between* **Sales** *and* **Products** *could be addressed*

```
http://host/service/$crossjoin(Products,Sales)?$filter=Products/ID eq Sales/ProductID
```

and would result in

```
{
  "@context": "http://host/service/$metadata#Collection(Edm.ComplexType)",
  "value": [
    {
      "Products@navigationLink": "Products(0)",
      "Sales@navigationLink": "Sales(42)",
    },
    {
      "Products@navigationLink": "Products(0)",
      "Sales@navigationLink": "Sales(57)",
    },
    …
    {
      "Products@navigationLink": "Products(99)",
      "Sales@navigationLink": "Sales(21)",
    }
  ]
}
```

4.16 Addressing All Entities in a Service

The symbolic resource **`$all`**, located at the service root, identifies the collection of all entities in a service, i.e. the union of all entity sets plus all singletons.

This symbolic resource is of type **`Collection(Edm.EntityType)`** and allows the **`$search`** system query option plus all other query options applicable to collections of entities.

The **`$all`** resource can be appended with a path segment containing the qualified name of an entity type in order to restrict the collections to entities of that type. Query options such as **`$select`**, **`$filter`**, **`$expand`**, and **`$orderby`** can be applied to this restricted set according to the specified type.

Example 49: all entities in a service that somehow match **`red`**

```
http://host/service/$all?$search=red
```

Example 50: all **`Customer`** *entities in a service whose name contains* **`red`**

```
http://host/service/$all/Model.Customer?$filter=contains(Name,'red')
```

4.17 Passing Query Options in the Request Body

The query options part of an OData URL can be quite long, potentially exceeding the maximum length of URLs supported by components involved in transmitting or processing the request. One way to avoid this is wrapping the request in a batch request, which has the penalty of needing to construct a well-formed batch request body.

An easier alternative for **`GET`** requests is to append **`/$query`** to the resource path of the URL, use the **`POST`** verb instead of **`GET`**, and pass the query options part of the URL in the request body.

Requests to paths ending in **`/$query`** MUST use the **`POST`** verb. Query options specified in the request body and query options specified in the request URL are processed together.

The request body MUST use a **`Content-Type`** of **`text/plain`**, **`application/x-www-form-urlencoded`**, or **`application/json`**.

For **`Content-Type: text/plain`**, the individual query options MUST be separated by **`&`** and MUST use the same percent-encoding as in URLs (especially: no spaces, tabs, or line breaks allowed) and MUST follow the syntax rules described in chapter 5.

Example 51: system query options in request body instead of URL

```
POST http://host/service/People/$query
Content-Type: text/plain

$filter=LastName%20eq%20'P%26G'&$select=FirstName,LastName
```

For **`Content-Type: application/x-www-form-urlencoded`**, the request body MUST be suitable *input* for the **`application/x-www-form-urlencoded`** parser in the **[URL Living Standard]**, section 5.1 such that the *output* of the parsing steps is the list of name/value pairs for the individual query options.

To guarantee this, clients are advised to make the request body the value of *output* after running the **`application/x-www-form-urlencoded`** serializer in the **[URL Living Standard]**, section 5.2 with *tuples* being the list of name/value pairs for the individual query options.

Example 52: The same payload as in example 51 can be sent with **`application/x-www-form-urlencoded`** *encoding. But the the* **`application/x-www-form-urlencoded`** *parser also accepts a different encoding:*

```
POST http://host/service/People/$query
Content-Type: application/x-www-form-urlencoded

%24filter=LastName+eq+%27P%26G%27&%24select=FirstName%2CLastName
```

This POST request would result from submitting the HTML form

```
<form method="post" action="http://host/service/People/$query"
      enctype="application/x-www-form-urlencoded">
  <input name="$filter" value="LastName eq 'P&G'">
  <input name="$select" value="FirstName,LastName">
</form>
```

which encodes spaces and ampersands (and more characters for which encoding is optional).

With **`Content-Type: application/json`** query options and function parameters are encoded in a request body that represents a JSON object. Its members include the individual query options. The name of a system query option MUST have the **`$`** prefix. The value MUST be

- a JSON number for **`$top`** and **`$skip`**, and
- a JSON string without percent-encoding for all other query options.

Example 53: The same request as in example 51 can be sent with **`application/json`** *encoding using the following payload:*

```
POST http://host/service/People/$query
Content-Type: application/json

{
  "$filter": "LastName eq 'P&G'",
  "$select": "FirstName,LastName"
}
```

Members of the JSON object also include parameters if the resource path is a function invocation or function import. In this case parameters MUST be represented like parameters in an action invocation **[OData-JSON, section 18]**, and in the resource path parentheses after the function name MUST be omitted.

Example 54: An employee's top ten leave requests from now to the end of the year pending their manager's approval.

```
POST http://host/service/Employees(23)/self.PendingLeaveRequests/$query
Content-Type: application/json

{
  "StartDate@expression": "now()",
  "EndDate": "2024-12-31",
```

```
  "Approver@expression": "Manager",
  "$top": 10
}
```

*The previous request looks analogous to a bound function invocation with expressions (like in example 31) if it is written using implicit parameter aliases (see **[OData-Protocol, section 11.5.4.1.1]**).*

```
GET http://host/service/Employees(23)/self.PendingLeaveRequests
  ?StartDate=now()
  &EndDate=2024-12-31
  &Approver=Manager
  &$top=10
```

5 Query Options

The query options part of an OData URL specifies three types of information: system query options, custom query options, and parameter aliases. All OData services MUST follow the query string parsing and construction rules defined in this section and its subsections.

5.1 System Query Options

System query options are query string parameters that control the amount and order of the data returned for the resource identified by the URL. The names of all system query options are optionally prefixed with a dollar (**$**) character. 4.01 Services MUST support case-insensitive system query option names specified with or without the **$** prefix. Clients that want to work with 4.0 services MUST use lower case names and specify the **$** prefix.

For **GET**, **PATCH**, and **PUT** requests the following rules apply:

- Resource paths identifying a single entity, a complex type instance, a collection of entities, or a collection of complex type instances allow **$compute**, **$expand** and **$select**.
- Resource paths identifying a collection allow **$filter**, **$search**, **$count**, **$orderby**, **$skip**, and **$top**.
- Resource paths ending in **/$count** allow **$filter** and **$search**.
- Resource paths not ending in **/$count** or **/$batch** allow **$format**.

For **POST** requests to an action URL the return type of the action determines the applicable system query options that a service MAY support, following the same rules as **GET** requests.

POST requests to an entity set follow the same rules as **GET** requests that return a single entity.

System query options SHOULD NOT be applied to a **DELETE** request.

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 reject any request that contains the unsupported option.

The same system query option, irrespective of casing or whether or not it is prefixed with a **$**, MUST NOT be specified more than once for any resource.

The semantics of all system query options are defined in **[OData-Protocol, section 11.2.1]**.

The grammar and syntax rules for system query options are defined in **[OData-ABNF]**.

Dynamic properties can be used in the same way as declared properties. If they are not defined on an instance, they evaluate to **null**.

5.1.1 Common Expression Syntax

The following operators, functions, and literals can be used in **$filter**, **$orderby**, and **$compute** expressions.

The **[OData-ABNF]** **commonExpr** syntax rule defines the formal grammar of common expressions.

The following subsections specify situations in which expressions evaluate to **null** if operands or parameters do not have the types expected by an operator or function. Notwithstanding these rules, if a service can determine such a type discrepancy for an expression that appears in a request independently of the underlying data, it MUST reject the request with an error message explaining the discrepancy. The determination can be based on, for example, the declared type of a property or the type of a literal value that occurs in the expression.

Example 55: In a search for people above a certain age

```
http://host/service/People?$filter=Age gt '50'
```

the expression would always evaluate to **null** *because the age 50 is erroneously given as a string and the* **$filter** *would return an empty result, although this is really the result of a typing error. That's why a "type mismatch" error must instead be returned in such a case.*

5.1.1.1 Logical Operators

OData defines a set of logical operators that evaluate to **true** or **false** (i.e. a **boolCommonExpr** as defined in **[OData-ABNF]**). Logical operators are typically used to filter a collection of resources.

The syntax rules for the logical operators are defined in **[OData-ABNF]**. 4.01 Services MUST support case-insensitive operator names. Clients that want to work with 4.0 services MUST use lower case operator names.

The six comparison operators can be used with all primitive values except **Edm.Binary**, **Edm.Stream**, and the **Edm.Geo** types. **Edm.Binary**, **Edm.Stream**, and the **Edm.Geo** types can only be compared to the **null** value using the **eq** and **ne** operators.

When applied to operands of numeric types, numeric promotion rules are applied.

The **eq**, **ne**, and **in** operators can be used with collection-valued operands, and the **eq** and **ne** operators can be used with operands of a structured type.

If at least one operand of an **eq**, **ne**, **lt**, **le**, **gt**, or **ge** operator is non-numeric and the operands have different types, the operator returns **null**.

The rules for the Boolean operators **and**, **or**, and **not** assume Boolean operands. If an operand of a Boolean operator is not Boolean, the operator returns **null**.

5.1.1.1.1 Equals

The **eq** operator returns **true** if the left operand is equal to the right operand, otherwise it returns **false**.

When applied to operands of entity types, the **eq** operator returns **true** if both operands represent the same entity, or both operands represent **null**.

When applied to operands of complex types, the **eq** operator returns **true** if both operands have the same structure and same values, or both operands represent **null**.

When applied to ordered collections, the **eq** operator returns **true** if both operands have the same cardinality and each member of the left operand is equal to the member of the right operand at the same index.

For services that support comparing unordered collections, the **eq** operator returns **true** if both operands are equal after applying the same ordering on both collections.

Each of the special values **null**, **-INF**, and **INF** is equal to itself, and only to itself.

The special value **NaN** is not equal to anything, even to itself.

5.1.1.1.2 Not Equals

The **ne** operator returns **true** if the left operand is not equal to the right operand, otherwise it returns **false**.

When applied to operands of entity types, the **ne** operator returns **true** if the two operands do not represent the same entity.

When applied to operands of complex types, the **ne** operator returns **true** if the operands do not have the same structure and same values.

When applied to ordered collections, the **ne** operator returns **true** if both operands do not have the same cardinality or any member of the left operand is not equal to the corresponding member of the right operand.

For services that support comparing unordered collections, the **ne** operator returns **true** if both operands do not have the same cardinality or do not contain the same members, in any order.

Each of the special values **null**, **-INF**, and **INF** is not equal to any value but itself.

The special value **NaN** is not equal to anything, even to itself.

The **null** value is not equal to any value but itself.

5.1.1.1.3 Greater Than

The **gt** operator returns **true** if the left operand is greater than the right operand, otherwise it returns **false**.

The special value **INF** is greater than any number, and any number is greater than **-INF**.

The Boolean value **true** is greater than **false**.

Services SHOULD order language-dependent strings according to the **Content-Language** of the response, and SHOULD annotate string properties with language-dependent order with the term **Core.IsLanguageDependent**, see **[OData-VocCore]**.

If any operand is **null**, the operator returns **false**.

5.1.1.1.4 Greater Than or Equal

The **ge** operator returns **true** if the left operand is greater than or equal to the right operand, otherwise it returns **false**.

See rules for **gt** and **eq** for details.

5.1.1.1.5 Less Than

The **lt** operator returns **true** if the left operand is less than the right operand, otherwise it returns **false**.

The special value **-INF** is less than any number, and any number is less than **INF**.

The Boolean value **false** is less than **true**.

Services SHOULD order language-dependent strings according to the **Content-Language** of the response, and SHOULD annotate string properties with language-dependent order with the term **Core.IsLanguageDependent**, see **[OData-VocCore]**.

If any operand is **null**, the operator returns **false**.

5.1.1.1.6 Less Than or Equal

The **le** operator returns **true** if the left operand is less than or equal to the right operand, otherwise it returns **false**.

See rules for **lt** and **eq** for details.

5.1.1.1.7 And

The **and** operator returns **true** if both the left and right operands evaluate to **true**, otherwise it returns **false**.

The **null** value is treated as unknown, so if one operand evaluates to **null** and the other operand to **false**, the **and** operator returns **false**. All other combinations with **null** return **null**.

5.1.1.1.8 Or

The **or** operator returns **false** if both the left and right operands both evaluate to **false**, otherwise it returns **true**.

The **null** value is treated as unknown, so if one operand evaluates to **null** and the other operand to **true**, the **or** operator returns **true**. All other combinations with **null** return **null**.

5.1.1.1.9 Not

The **not** operator returns **true** if the operand returns **false**, otherwise it returns **false**.

The **null** value is treated as unknown, so **not null** returns **null**.

5.1.1.1.10 Has

The **has** operator returns **true** if the right operand is an enumeration value whose flag(s) are set on the left operand.

The **null** value is treated as unknown, so if one operand evaluates to **null**, the **has** operator returns **null**.

5.1.1.1.11 In

The **in** operator returns **true** if the equality comparison of the left operand with at least one member of the right operand returns **true**. The right operand MUST be either a comma-separated list of zero or more primitive values, enclosed in parentheses, or a single expression that resolves to a collection. If the right operand is an empty collection or list of values, the **in** operator returns **false**.

5.1.1.1.12 Logical Operator Examples

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

Example 56: all products with a **Name** *equal to* **Milk**

```
http://host/service/Products?$filter=Name eq 'Milk'
```

Example 57: all products with a **Name** *not equal to* **Milk**

```
http://host/service/Products?$filter=Name ne 'Milk'
```

Example 58: all products with a **Name** *greater than* **Milk**:

```
http://host/service/Products?$filter=Name gt 'Milk'
```

Example 59: all products with a **Name** *greater than or equal to* **Milk**:

```
http://host/service/Products?$filter=Name ge 'Milk'
```

Example 60: all products with a **Name** *less than* **Milk**:

```
http://host/service/Products?$filter=Name lt 'Milk'
```

Example 61: all products with a **Name** *less than or equal to* **Milk**:

```
http://host/service/Products?$filter=Name le 'Milk'
```

Example 62: all products with a **Name** *equal to* **Milk** *that also have a* **Price** *less than 2.55:*

```
http://host/service/Products?$filter=Name eq 'Milk' and Price lt 2.55
```

Example 63: all products that either have a **`Name`** *equal to* **`Milk`** *or have a* **`Price`** *less than 2.55:*

```
http://host/service/Products?$filter=Name eq 'Milk' or Price lt 2.55
```

Example 64: all products that do not have a **`Name`** *that ends with* **`ilk`***:*

```
http://host/service/Products?$filter=not endswith(Name,'ilk')
```

Example 65: all products whose **`style`** *value includes* **`Yellow`***:*

```
http://host/service/Products?$filter=style has Sales.Pattern'Yellow'
```

Example 66: all products whose **`Name`** *is* **`Milk`** *or* **`Cheese`***:*

```
http://host/service/Products?$filter=Name in ('Milk', 'Cheese')
```

5.1.1.2 Arithmetic Operators

OData defines a set of arithmetic operators that require operands that evaluate to numeric types. Arithmetic operators are typically used to filter a collection of resources. However, services MAY allow using arithmetic operators with the **`$orderby`** system query option.

If an operand of an arithmetic operator is **`null`** or has a non-allowed type, the result is **`null`**.

The syntax rules for the arithmetic operators are defined in **[OData-ABNF]**. 4.01 Services MUST support case-insensitive operator names. Clients that want to work with 4.0 services MUST use lower case operator names.

5.1.1.2.1 Addition

The **`add`** operator adds the left and right numeric operands.

For operands of type **`Edm.Decimal`** the scale of the result is $\text{scaleof}(A \texttt{ add } B) = \max(\text{scaleof}(A), \text{scaleof}(B))$, or **`variable`** if any operand has variable scale.

The **`add`** operator is also valid for the following time-related operands:

- **`DateTimeOffset add Duration`** results in a **`DateTimeOffset`**
- **`Duration add Duration`** results in a **`Duration`**
- **`Date add Duration`** results in a **`Date`**

The rules for time-related operands are defined in **[XML-Schema-2]**, section E.3.3. Specifically, for adding a duration to a date:

- Convert date to datetime (in any timezone) with a zero time component
- Add/subtract duration
- Convert to date by removing the time and timezone components

Thus, today plus a positive duration smaller than one day is today; today minus a positive duration smaller than one day is yesterday.

5.1.1.2.2 Subtraction

The **`sub`** operator subtracts the right numeric operand from the left numeric operand.

For operands of type **Edm.Decimal** the scale of the result is $\text{scaleof}(A \texttt{ sub } B) = \max(\text{scaleof}(A), \text{scaleof}(B))$, or **variable** if any operand has variable scale.

The **sub** operator is also valid for the following time-related operands:

- **DateTimeOffset sub Duration** results in a **DateTimeOffset**
- **Duration sub Duration** results in a **Duration**
- **DateTimeOffset sub DateTimeOffset** results in a **Duration**
- **Date sub Duration** results in a **Date**
- **Date sub Date** results in a **Duration**

The rules for time-related operands are defined in **[XML-Schema-2]**, section E.3.3. Specifically for subtracting a duration from a date see the preceding section.

5.1.1.2.3 Negation

The negation operator, represented by a minus (**-**) sign, changes the sign of its numeric or **Duration** operand.

5.1.1.2.4 Multiplication

The **mul** operator multiplies the left and right numeric operands. The **mul** operator is also valid for multiplying a **Duration** value with a numeric value.

For operands of type **Edm.Decimal** the scale of the result is $\text{scaleof}(A \texttt{ mul } B) = \text{scaleof}(A) + \text{scaleof}(B)$, **floating** if any operand has floating scale, or else **variable** if any operand has variable scale.

5.1.1.2.5 Division

The **div** and **divby** operators divide the left numeric operand by the right numeric operand. They are also valid for dividing a **Duration** value by a numeric value.

If the left operand is of type **Edm.Decimal** with floating scale, **Edm.Double**, or **Edm.Single**, then positive **div** zero returns **INF**, negative **div** zero returns **-INF**, and zero **div** zero returns **NaN**. For all other types the request fails.

For operands of type **Edm.Decimal** the result is computed with maximal decimal scale. If any operand has floating scale, the result has floating scale, else if any operand has variable scale, the result has variable scale. Otherwise the resulting scale is service-specific, and clients can use **cast** to force the result to a specific scale.

The **div** and **divby** operators differ in their handling of integers. If both operands to **div** are of an integer type, the result is an integer representing the whole number of times the right operator fits into the left operator. The **divby** operator, on the other hand, promotes both operands to decimal before computing the result, may yield a fractional result, and does not fail for **divby** zero, returning **-INF**, **INF**, or **NaN** depending on the sign of the left operand.

5.1.1.2.6 Modulo

The **mod** operator returns the remainder when the left numeric operand is divided by the right numeric operand. The sign of the result is the same as the sign of the left operand. If the right operand is zero, the request fails.

For operands of type **Edm.Decimal** the scale of the result is $\text{scaleof}(A \texttt{ mod } B) = \max(\text{scaleof}(A), \text{scaleof}(B))$, or **variable** if any operand has variable scale.

5.1.1.2.7 Arithmetic Operator Examples

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

Example 67: all products with a Price of 2.55:

```
http://host/service/Products?$filter=Price add 2.45 eq 5.00
```

Example 68: all products with a Price of 2.55:

```
http://host/service/Products?$filter=Price sub 0.55 eq 2.00
```

Example 69: all products with a Price of 2.55:

```
http://host/service/Products?$filter=Price mul 2.0 eq 5.10
```

Example 70: all products with a Price of 2.55:

```
http://host/service/Products?$filter=Price div 2.55 eq 1
```

Example 71: all products with an integer Rating value of 4 or 5:

```
http://host/service/Products?$filter=Rating div 2 eq 2
```

Example 72: all products with an integer Rating value of 5:

```
http://host/service/Products?$filter=Rating divby 2 eq 2.5
```

Example 73: all products with a Rating exactly divisible by 5:

```
http://host/service/Products?$filter=Rating mod 5 eq 0
```

5.1.1.3 Grouping

The Grouping operator (open and close parenthesis "**( )**") controls the evaluation order of an expression. The Grouping operator returns the expression grouped inside the parenthesis.

Example 74: all products because 9 mod 3 is 0

```
http://host/service/Products?$filter=(4 add 5) mod (4 sub 1) eq 0
```

5.1.1.4 Canonical Functions

In addition to operators, a set of functions is also defined for use with the **$compute**, **$filter** or **$orderby** system query options, or in parameter alias values. The following sections 5.1.1.5.1 to 5.1.1.11.3 describe the available functions. The **case** and lambda operators have a slightly different syntax.

Note: ISNULL or COALESCE operators are not defined. Instead, OData defines a **null** literal that can be used in comparisons.

If a parameter of a canonical function is **null**, the function returns **null**. If the types of parameters do not match the function signature, the function also returns **null**.

The syntax rules for all functions are defined in **[OData-ABNF]**. 4.01 Services MUST support case-insensitive canonical function names. Clients that want to work with 4.0 services MUST use lower case canonical function names.

5.1.1.5 String and Collection Functions

5.1.1.5.1 concat

The **concat** function has three overloads, with the following signatures:

```
Edm.String concat(Edm.String,Edm.String)
Collection concat(Collection,Collection)
```

```
OrderedCollection concat(OrderedCollection,OrderedCollection)
```

The **concat** function with string parameter values returns a string that appends the second parameter string value to the first.

The concat function with collection parameter values returns a collection that appends all items of the second collection to the first. If both collections are ordered, the result is also ordered.

The **concatMethodCallExpr** syntax rule defines how the **concat** function is invoked.

Example 75: all customers from Berlin, Germany

```
http://host/service/Customers?$filter=concat(concat(City,', '),Country) eq 'Berlin, Germany'
```

5.1.1.5.2 contains

The **contains** function has two overloads, with the following signatures:

```
Edm.Boolean contains(Edm.String,Edm.String)
Edm.Boolean contains(OrderedCollection,OrderedCollection)
```

The **contains** function with string parameter values returns **true** if the second string is a substring of the first string, otherwise it returns **false**. String comparison is case-sensitive, case-insensitive comparison can be achieved in combination with **tolower** or **toupper**.

The **contains** function with ordered collection parameter values returns **true** if the first collection can be transformed into the second collection by removing zero or more items from the beginning or the end of the first collection.

The **containsMethodCallExpr** syntax rule defines how the **contains** function is invoked.

Example 76: all customers with a **CompanyName** *that contains* **Alfreds**

```
http://host/service/Customers?$filter=contains(CompanyName,'Alfreds')
```

5.1.1.5.3 endswith

The **endswith** function has two overloads, with the following signatures:

```
Edm.Boolean endswith(Edm.String,Edm.String)
Edm.Boolean endswith(OrderedCollection,OrderedCollection)
```

The **endswith** function with string parameter values returns **true** if the first string ends with the second string, otherwise it returns **false**. String comparison is case-sensitive, case-insensitive comparison can be achieved in combination with **tolower** or **toupper**.

The **endswith** function with ordered collection parameter values returns **true** if the first collection can be transformed into the second collection by removing zero or more items from the beginning of the first collection.

The **endsWithMethodCallExpr** syntax rule defines how the **endswith** function is invoked.

Example 77: all customers with a **CompanyName** *that ends with* **Futterkiste**

```
http://host/service/Customers?$filter=endswith(CompanyName,'Futterkiste')
```

5.1.1.5.4 indexof

The **indexof** function has two overloads, with the following signatures:

```
Edm.Int32 indexof(Edm.String,Edm.String)
Edm.Int32 indexof(OrderedCollection,OrderedCollection)
```

The **indexof** function with string parameter values returns the zero-based character position of the first occurrence of the second string in the first string, or -1 if the first string does not contain the second string. String comparison is case-sensitive, case-insensitive comparison can be achieved in combination with **tolower** or **toupper**.

The **indexof** function with ordered collection parameter values returns the zero-based index of the first occurrence of the second collection in the first collection, or -1 if the first collection does not contain the second collection.

The **indexOfMethodCallExpr** syntax rule defines how the **indexof** function is invoked.

Example 78: all customers with a **CompanyName** *containing* **lfreds** *starting at the second character*

```
http://host/service/Customers?$filter=indexof(CompanyName,'lfreds') eq 1
```

5.1.1.5.5 length

The **length** function has two overloads, with the following signatures:

```
Edm.Int32 length(Edm.String)
Edm.Int32 length(Collection)
```

The **length** function with a string parameter value returns the number of characters in the string.

The **length** function with a collection parameter value returns the number of itens in the collection.

The **lengthMethodCallExpr** syntax rule defines how the **length** function is invoked.

Example 79: all customers with a **CompanyName** *that is 19 characters long*

```
http://host/service/Customers?$filter=length(CompanyName) eq 19
```

5.1.1.5.6 startswith

The **startswith** function has two overloads, with the following signatures:

```
Edm.Boolean startswith(Edm.String,Edm.String)
Edm.Boolean startswith(Collection,Collection)
```

The **startswith** function with string parameter values returns **true** if the first string starts with the second string, otherwise it returns **false**. String comparison is case-sensitive, case-insensitive comparison can be achieved in combination with **tolower** or **toupper**.

The **startswith** function with ordered collection parameter values returns **true** if the first collection can be transformed into the second collection by removing zero or more items from the end of the first collection.

The **startsWithMethodCallExpr** syntax rule defines how the **startswith** function is invoked.

Example 80: all customers with a **CompanyName** *that starts with* **Alfr**

```
http://host/service/Customers?$filter=startswith(CompanyName,'Alfr')
```

5.1.1.5.7 substring

The **substring** function has four overloads, with the following signatures:

```
Edm.String substring(Edm.String,Edm.Int32)
Edm.String substring(Edm.String,Edm.Int32,Edm.Int32)
OrderedCollection substring(OrderedCollection,Edm.Int32)
OrderedCollection substring(OrderedCollection,Edm.Int32,Edm.Int32)
```

The two-parameter **substring** function with string parameter values returns a substring of the first parameter string value, starting at the Nth character and finishing at the last character (where N is the second parameter integer value). The three-parameter **substring** function with string parameter values returns a substring of the first parameter string value identified by selecting up to M characters starting at the Nth character (where N is the second parameter integer value and M is the third parameter integer value).

The two-parameter **substring** function with ordered collection parameter values returns an ordered collection consisting of all items of the first collection starting at the Nth item and finishing at the last item. The three-parameter **substring** function with ordered collection parameter values returns an ordered collection consisting of up to M items of the first collection starting at the Nth item (where N is the second parameter integer value and M is the third parameter integer value).

The start index N is zero-based.

If the start index N is larger than the length of the string/collection, an empty string/collection is returned.

If the length M is larger than the length of the remaining string/collection starting at the Nth character/item, as many characters/items as are available are returned.

A negative length M is a bad request.

A negative start index N, if supported, returns a string/collection starting N characters/items before the end of the string/collection.

The **substringMethodCallExpr** syntax rule defines how the **substring** function is invoked.

Example 81: all customers with a **CompanyName** *of* **lfreds Futterkiste** *once the first character has been removed*

```
http://host/service/Customers?$filter=substring(CompanyName,1) eq 'lfreds Futterkiste'
```

Example 82: all customers with a **CompanyName** *that has* **lf** *as the second and third characters, e.g,* **Alfreds Futterkiste**

```
http://host/service/Customers?$filter=substring(CompanyName,1,2) eq 'lf'
```

5.1.1.6 Collection Functions

5.1.1.6.1 hassubset

The **hassubset** function has the following signature:

```
Edm.Boolean hassubset(Collection, Collection)
```

The **hassubset** function returns **true** if the first collection can be transformed into the second collection by reordering and/or removing zero or more items. The **hasSubsetMethodCallExpr** syntax rule defines how the **hassubset** function is invoked.

Example 83: **hassubset** *expressions that return* **true**

```
hassubset([4,1,3],[4,1,3])
```

```
hassubset([4,1,3],[1,3,4])
```

```
hassubset([4,1,3],[3,1])
```

```
hassubset([4,1,3],[4,3])
```

```
hassubset([4,1,3,1],[1,1])
```

Example 84: **hassubset** *expression that returns* **false***:* **1** *appears only once in the left operand*

```
hassubset([1,2],[1,1,2])
```

5.1.1.6.2 hassubsequence

The **hassubsequence** function has the following signature:

```
Edm.Boolean hassubsequence(OrderedCollection,OrderedCollection)
```

The **hassubsequence** function returns **true** if the first collection can be transformed into the second collection by removing zero or more items. The **hasSubsequenceMethodCallExpr** syntax rule defines how the **hassubsequence** function is invoked.

Example 85: **hassubsequence** *expressions that return* **true**

```
hassubsequence([4,1,3],[4,1,3])
```

```
hassubsequence([4,1,3],[4,1])
```

```
hassubsequence([4,1,3],[4,3])
```

```
hassubsequence([4,1,3,1],[1,1])
```

Example 86: **hassubsequence** *expressions that return* **false**

```
hassubsequence([4,1,3],[1,3,4])
```

```
hassubsequence([4,1,3],[3,1])
```

```
hassubsequence([1,2],[1,1,2])
```

5.1.1.7 String Functions

5.1.1.7.1 matchespattern

The **matchespattern** function has the following signatures:

```
Edm.Boolean matchespattern(Edm.String,Edm.String)
Edm.Boolean matchespattern(Edm.String,Edm.String,Edm.String)
```

The second parameter MUST evaluate to a string containing an **[ECMAScript]** (JavaScript) regular expression, otherwise the function returns **null**. The **matchespattern** function returns **true** if the first parameter evaluates to a string matching that regular expression, using syntax and semantics of ECMAScript regular expressions, otherwise

it returns **false**. If the optional third parameter is provided, it MUST evaluate to a string consisting of ECMAScript regular expression flags to modify the match, otherwise the function returns **null**.

Example 87: all customers with a **CompanyName** *that match the (percent-encoded) regular expression* **^A.*e$**

```
http://host/service/Customers?$filter=matchespattern(CompanyName,'%5EA.*e$')
```

Example 88: all customers with a **FormattedAddress** *that contains a line ending with* **berg** *or ends with* **berg**

```
http://host/service/Customers?$filter=matchespattern(FormattedAddress,'berg$','m')
```

5.1.1.7.2 tolower

The **tolower** function has the following signature:

```
Edm.String tolower(Edm.String)
```

The **tolower** function returns the input parameter string value with all uppercase characters converted to lowercase according to Unicode rules. The **toLowerMethodCallExpr** syntax rule defines how the **tolower** function is invoked.

Example 89: all customers with a **CompanyName** *that equals* **alfreds futterkiste** *once any uppercase characters have been converted to lowercase*

```
http://host/service/Customers?$filter=tolower(CompanyName) eq 'alfreds futterkiste'
```

5.1.1.7.3 toupper

The **toupper** function has the following signature:

```
Edm.String toupper(Edm.String)
```

The **toupper** function returns the input parameter string value with all lowercase characters converted to uppercase according to Unicode rules. The **toUpperMethodCallExpr** syntax rule defines how the **toupper** function is invoked.

Example 90: all customers with a **CompanyName** *that equals* **ALFREDS FUTTERKISTE** *once any lowercase characters have been converted to uppercase*

```
http://host/service/Customers?$filter=toupper(CompanyName) eq 'ALFREDS FUTTERKISTE'
```

5.1.1.7.4 trim

The **trim** function has the following signature:

```
Edm.String trim(Edm.String)
```

The **trim** function returns the input parameter string value with all leading and trailing whitespace characters, according to Unicode rules, removed. The **trimMethodCallExpr** syntax rule defines how the **trim** function is invoked.

Example 91: all customers with a **CompanyName** *without leading or trailing whitespace characters*

```
http://host/service/Customers?$filter=trim(CompanyName) eq CompanyName
```

5.1.1.8 Date and Time Functions

5.1.1.8.1 date

The **date** function has the following signature:

```
Edm.Date date(Edm.DateTimeOffset)
```

The **date** function returns the date part of the **DateTimeOffset** parameter value, evaluated in the time zone of the **DateTimeOffset** parameter value.

5.1.1.8.2 day

The **day** function has the following signatures:

```
Edm.Int32 day(Edm.Date)
Edm.Int32 day(Edm.DateTimeOffset)
```

The **day** function returns the day component **Date** or **DateTimeOffset** parameter value, evaluated in the time zone of the **DateTimeOffset** parameter value. The **dayMethodCallExpr** syntax rule defines how the **day** function is invoked.

Services that are unable to preserve the offset of **Edm.DateTimeOffset** values and instead normalize the values to some common time zone (for example UTC) MUST fail evaluation of the **day** function for literal **Edm.DateTimeOffset** values that are not stated in the time zone of the normalized values.

Example 92: all employees born on the 8th day of a month

```
http://host/service/Employees?$filter=day(BirthDate) eq 8
```

5.1.1.8.3 fractionalseconds

The **fractionalseconds** function has the following signatures:

```
Edm.Decimal fractionalseconds[(Edm.DateTimeOffset)
Edm.Decimal fractionalseconds(Edm.TimeOfDay)
```

The **fractionalseconds** function returns the fractional seconds component of the **DateTimeOffset** or **TimeOfDay** parameter value as a non-negative decimal value less than 1. The **fractionalsecondsMethodCallExpr** syntax rule defines how the **fractionalseconds** function is invoked.

Example 93: all employees born less than 100 milliseconds after a full second of any minute of any hour on any day

```
http://host/service/Employees?$filter=[fractionalseconds(BirthDate) lt 0.1
```

5.1.1.8.4 hour

The **hour** function has the following signatures:

```
Edm.Int32 hour(Edm.DateTimeOffset)
Edm.Int32 hour(Edm.TimeOfDay)
```

The **hour** function returns the hour component (**0** to **23**) of the **DateTimeOffset** or **TimeOfDay** parameter value, evaluated in the time zone of the **DateTimeOffset** parameter value. The **hourMethodCallExpr** syntax rule defines how the **hour** function is invoked.

Services that are unable to preserve the offset of **Edm.DateTimeOffset** values and instead normalize the values to some common time zone (for example UTC) MUST fail evaluation of the **hour** function for literal **Edm.DateTimeOffset** values that are not stated in the time zone of the normalized values.

Example 94: all employees born in hour 4, between 04:00 (inclusive) and 05:00 (exclusive)

```
http://host/service/Employees?$filter=hour(BirthDate) eq 4
```

5.1.1.8.5 maxdatetime

The **maxdatetime** function has the following signature:

```
Edm.DateTimeOffset maxdatetime()
```

The **maxdatetime** function returns the latest possible point in time as a **DateTimeOffset** value.

5.1.1.8.6 mindatetime

The **mindatetime** function has the following signature:

```
Edm.DateTimeOffset mindatetime()
```

The **mindatetime** function returns the earliest possible point in time as a **DateTimeOffset** value.

5.1.1.8.7 minute

The **minute** function has the following signatures:

```
Edm.Int32 minute(Edm.DateTimeOffset)
Edm.Int32 minute(Edm.TimeOfDay)
```

The **minute** function returns the minute component (**0** to **59**) of the **DateTimeOffset** or **TimeOfDay** parameter value, evaluated in the time zone of the **DateTimeOffset** parameter value. The **minuteMethodCallExpr** syntax rule defines how the **minute** function is invoked.

Example 95: all employees born in minute 40 of any hour on any day

```
http://host/service/Employees?$filter=minute(BirthDate) eq 40
```

5.1.1.8.8 month

The **month** function has the following signatures:

```
Edm.Int32 month(Edm.Date)
Edm.Int32 month(Edm.DateTimeOffset)
```

The **month** function returns the month component of the **Date** or **DateTimeOffset** parameter value, evaluated in the time zone of the **DateTimeOffset** parameter value. The **monthMethodCallExpr** syntax rule defines how the **month** function is invoked.

Services that are unable to preserve the offset of **Edm.DateTimeOffset** values and instead normalize the values to some common time zone (for example UTC) MUST fail evaluation of the **month** function for literal **Edm.DateTimeOffset** values that are not stated in the time zone of the normalized values.

Example 96: all employees born in May

```
http://host/service/Employees?$filter=month(BirthDate) eq 5
```

5.1.1.8.9 now

The **now** function has the following signature:

```
Edm.DateTimeOffset now()
```

The **now** function returns the current point in time (date and time with time zone) as a **DateTimeOffset** value.

Services are free to choose the time zone for the current point, for example UTC. Services that are unable to preserve the offset of **Edm.DateTimeOffset** values and instead normalize the values to some common time zone SHOULD return a value in the normalized time zone (for example UTC).

5.1.1.8.10 second

The **second** function has the following signatures:

```
Edm.Int32 second(Edm.DateTimeOffset)
Edm.Int32 second(Edm.TimeOfDay)
```

The **second** function returns the second component (**0** to **59** for regular seconds, and **60** for leap seconds, without the fractional part) of the **DateTimeOffset** or **TimeOfDay** parameter value. The **secondMethodCallExpr** syntax rule defines how the **second** function is invoked.

Example 97: all employees born in second 40 of any minute of any hour on any day

```
http://host/service/Employees?$filter=second(BirthDate) eq 40
```

5.1.1.8.11 time

The **time** function has the following signature:

```
Edm.TimeOfDay time(Edm.DateTimeOffset)
```

The **time** function returns the time part of the **DateTimeOffset** parameter value, evaluated in the time zone of the **DateTimeOffset** parameter value.

Services that are unable to preserve the offset of **Edm.DateTimeOffset** values and instead normalize the values to some common time zone (for example UTC) MUST fail evaluation of the **time** function for literal **Edm.DateTimeOffset** values that are not stated in the time zone of the normalized values.

5.1.1.8.12 totaloffsetminutes

The **totaloffsetminutes** function has the following signature:

```
Edm.Int32 totaloffsetminutes(Edm.DateTimeOffset)
```

The **totaloffsetminutes** function returns the signed number of minutes in the time zone offset part of the **DateTimeOffset** parameter value, evaluated in the time zone of the **DateTimeOffset** parameter value.

5.1.1.8.13 totalseconds

The **totalseconds** function has the following signature:

```
Edm.Decimal totalseconds(Edm.Duration)
```

The **totalseconds** function returns the duration of the value in total seconds, including fractional seconds.

5.1.1.8.14 year

The **year** function has the following signatures:

```
Edm.Int32 year(Edm.Date)
Edm.Int32 year(Edm.DateTimeOffset)
```

The **year** function returns the year component of the **Date** or **DateTimeOffset** parameter value, evaluated in the time zone of the **DateTimeOffset** parameter value. The **yearMethodCallExpr** syntax rule defines how the **year** function is invoked.

Services that are unable to preserve the offset of **Edm.DateTimeOffset** values and instead normalize the values to some common time zone (for example UTC) MUST fail evaluation of the **year** function for literal **Edm.DateTimeOffset** values that are not stated in the time zone of the normalized values.

Example 98: all employees born in 1971

```
http://host/service/Employees?$filter=year(BirthDate) eq 1971
```

5.1.1.9 Arithmetic Functions

5.1.1.9.1 ceiling

The **ceiling** function has the following signatures

```
Edm.Double ceiling(Edm.Double)
Edm.Decimal ceiling(Edm.Decimal)
```

The **ceiling** function rounds the input numeric parameter up to the nearest numeric value with no decimal component. The **ceilingMethodCallExpr** syntax rule defines how the **ceiling** function is invoked.

Example 99: all orders with freight costs that round up to 32

```
http://host/service/Orders?$filter=ceiling(Freight) eq 32
```

5.1.1.9.2 floor

The **floor** function has the following signatures

```
Edm.Double floor(Edm.Double)
Edm.Decimal floor(Edm.Decimal)
```

The **floor** function rounds the input numeric parameter down to the nearest numeric value with no decimal component. The **floorMethodCallExpr** syntax rule defines how the **floor** function is invoked.

Example 100: all orders with freight costs that round down to 32

```
http://host/service/Orders?$filter=floor(Freight) eq 32
```

5.1.1.9.3 round

The **round** function has the following signatures

```
Edm.Double round(Edm.Double)
Edm.Decimal round(Edm.Decimal)
```

The **round** function rounds the input numeric parameter to the nearest numeric value with no decimal component. The mid-point between two integers is rounded away from zero, i.e. 0.5 is rounded to 1 and -0.5 is rounded to -1. The **roundMethodCallExpr** syntax rule defines how the **round** function is invoked.

Example 101: all orders with freight costs that round to 32

```
http://host/service/Orders?$filter=round(Freight) eq 32
```

5.1.1.10 Type Functions

5.1.1.10.1 cast

The **cast** function has the following signatures:

```
type cast(type)
type cast(expression,type)
```

The single parameter **cast** function returns the current instance cast to the type specified. The two-parameter **cast** function returns the object referred to by the expression cast to the type specified.

The **cast** function follows these assignment rules:

1. The **null** value can be cast to any type.
2. Primitive types are cast to **Edm.String** or a type definition based on it by using the literal representation used in payloads, and WKT (well-known text) format for **Geo** types, see rules **fullCollectionLiteral**, **fullLineStringLiteral**, **fullMultiPointLiteral**, **fullMultiLineStringLiteral**, **fullMultiPolygonLiteral**, **fullPointLiteral**, and **fullPolygonLiteral** in **[OData-ABNF]**. The cast fails if the target type specifies an insufficient **MaxLength**.
3. **Edm.String**, or a type definition based on **Edm.String**, can be cast to a primitive type if the string contains a literal representation for the target type.
4. Numeric primitive types are cast to each other with appropriate rounding. The cast fails if the integer part doesn't fit into the target type.
5. **Edm.DateTimeOffset**, **Edm.Duration**, and **Edm.TimeOfDay** values can be cast to the same type with a different precision with appropriate rounding.
6. Non-Unicode string values can be cast to a Unicode string type definition. Unicode string values can be cast to a non-Unicode string type definition if the Unicode string only contains ASCII characters.
7. Structured types are assignable to their type or a direct or indirect base type.
8. Collections are cast item by item.
9. Enumeration types are cast to integer types based on the numeric value of the enumeration member. The cast fails if the numeric value is not in the value range of the target type.
10. Integer types are cast to enumeration types based on the numeric value of the enumeration members of the target type. For non-flag enumeration types the cast fails if there is no enumeration member with the same numeric value as the integer value. For flag enumeration types the cast fails if the integer value is not in the value range of the underlying integer type of the enumeration type.
11. String values containing a representation of a date-time value according to **[XML-Schema-2]**, section 3.3.7 dateTime, can be cast to **Edm.DateTimeOffset**. If the string value does not contain a time-zone offset, it is treated as UTC.

If the cast fails, the **cast** function returns **null**.

5.1.1.10.2 isof

The **isof** function has the following signatures

```
Edm.Boolean isof(type)
Edm.Boolean isof(expression,type)
```

The single parameter **isof** function returns **true** if the current instance is assignable to the type specified, according to the assignment rules for the **cast** function, otherwise it returns **false**.

The two parameter **isof** function returns **true** if the object referred to by the expression is assignable to the type specified, according to the same rules, otherwise it returns **false**.

The **isofExpr** syntax rule defines how the **isof** function is invoked.

Example 102: orders that are also **BigOrders**

```
http://host/service/Orders?$filter=isof(NorthwindModel.BigOrder)
```

```
http://host/service/Orders?$filter=isof($it,NorthwindModel.BigOrder)
```

Example 103: orders of a customer that is a **VIPCustomer**

```
http://host/service/Orders?$filter=isof(Customer,NorthwindModel.VIPCustomer)
```

5.1.1.11 Geo Functions

5.1.1.11.1 geo.distance

The **geo.distance** function has the following signatures:

```
Edm.Double geo.distance(Edm.GeographyPoint,Edm.GeographyPoint)
Edm.Double geo.distance(Edm.GeometryPoint,Edm.GeometryPoint)
```

The **geo.distance** function returns the shortest distance between the two points in the coordinate reference system signified by the two points' SRIDs.

5.1.1.11.2 geo.intersects

The **geo.intersects** function has the following signatures:

```
Edm.Boolean geo.intersects(Edm.GeographyPoint,Edm.GeographyPolygon)
Edm.Boolean geo.intersects(Edm.GeometryPoint,Edm.GeometryPolygon)
```

The **geo.intersects** function returns **true** if the specified point lies within the interior or on the boundary of the specified polygon, otherwise it returns **false**.

5.1.1.11.3 geo.length

The **geo.length** function has the following signatures:

```
Edm.Double geo.length(Edm.GeographyLineString)
Edm.Double geo.length(Edm.GeometryLineString)
```

The **geo.length** function returns the total length of its line string parameter in the coordinate reference system signified by its SRID.

5.1.1.12 Conditional Operators

5.1.1.12.1 `case`

The **`case`** operator has a comma-separated lists of arguments:

```
expression case(Edm.Boolean:expression, …, Edm.Boolean:expression)
```

Each argument is a pair of expressions separated by a colon (**`:`**), where the first expression — the condition — MUST be a Boolean expression, and the second expression — the result — may evaluate to any type.

The **`case`** operator evaluates the condition in each pair, starting with the leftmost pair, and stops as soon as a condition evaluates to **`true`**. It then returns the value of the result of this pair. It returns **`null`** if none of the conditions in any pair evaluates to **`true`**. Clients can specify a last pair whose condition is **`true`** to get a non-**`null`** "default/else/otherwise" result.

Boolean expressions containing **`DateTimeOffset`** or **`TimeOfDay`** literals without the optional seconds part will introduce ambiguity for parsers. Clients SHOULD use whitespace or parentheses to avoid ambiguity.

Clients SHOULD ensure that the results in all pairs are compatible. If all results are of the same type, the type of the **`case`** expression is of that type. If all results are of numeric type, then the type of the **`case`** expression is a numeric type capable of representing any of these expressions according to standard type promotion rules.

Services MAY support **`case`** expressions containing parameters of incompatible types, in which case the case expression is treated as **`Edm.Untyped`** and its value has the type of the parameter expression selected by the case statement.

Example 104: compute signum(X)

```
$compute=case(X gt 0:1,X lt 0:-1,true:0) as SignumX
```

5.1.1.13 Lambda Operators

OData defines two operators that evaluate a Boolean expression on a collection. Both must be prepended with a path expression that identifies a collection.

4.01 Services MUST support case-insensitive lambda operator names. Clients that want to work with 4.0 services MUST use lower case lambda operator names.

The argument of a lambda operator is a case-sensitive lambda variable name followed by a colon (**`:`**) and a Boolean expression that uses the lambda variable name to refer to properties of the instance or of members of the collection identified by the path expression.

If the name chosen for the lambda variable matches a property name of the current resource referenced by the resource path, the lambda variable takes precedence. Clients can prefix properties of the current resource referenced by the resource path with **`$it`**.

Other path expressions in the Boolean expression neither prefixed with the lambda variable nor **`$it`** are evaluated in the scope of the instance or of members of the collection at the origin of the path expression prepended to the lambda operator.

5.1.1.13.1 `any`

The **`any`** operator applies a Boolean expression to each member of a collection and returns **`true`** if and only if the expression is **`true`** for any member of the collection, otherwise it returns **`false`**. This implies that the **`any`** operator always returns **`false`** for an empty collection.

The **any** operator can be used without an argument expression. This short form returns **false** if and only if the collection is empty.

Example 105: all **Orders** *that have any* **Items** *with a* **Quantity** *greater than* **100**

```
http://host/service/Orders?$filter=Items/any(d:d/Quantity gt 100)
```

Example 106: all customers having an order with a deviating shipping address. The **Address** *in the argument expression is evaluated in the scope of the* **Customers** *collection.*

```
http://host/service/Customers?$filter=Orders/any(o:o/ShippingAddress ne Address)
```

Example 107: all categories along with their products used in some order with a deviating unit price. The unprefixed **UnitPrice** *in the argument expression is evaluated in the scope of the expanded* **Products***.*

```
http://host/service/Categories?$expand=Products(
  $filter=OrderItems/any(oi:oi/UnitPrice ne UnitPrice))
```

5.1.1.13.2 all

The **all** operator applies a Boolean expression to each member of a collection and returns **true** if the expression is **true** for all members of the collection, otherwise it returns **false**. This implies that the **all** operator always returns **true** for an empty collection.

The **all** operator cannot be used without an argument expression.

Example 108: all **Orders** *that have only* **Items** *with a* **Quantity** *greater than* **100**

```
http://host/service/Orders?$filter=Items/all(d:d/Quantity gt 100)
```

5.1.1.14 Literals

5.1.1.14.1 Primitive Literals

Primitive literals can appear in the resource path as key property values, and in the query part, for example, as operands in **$filter** expressions. They are represented according to the **primitiveLiteral** rule in **[OData-ABNF]**. The interpretation of a **timeOfDayLiteral** in which the **second** is omitted is not defined by this specification. For maximum interoperability, senders SHOULD always include the **second**.

Example 109: expressions using primitive literals

```
NullValue eq null
```

```
TrueValue eq true
```

```
FalseValue eq false
```

```
Custom.Base64UrlDecode(binary'T0RhdGE') eq 'OData'
```

```
IntegerValue lt -128
```

```
DoubleValue ge 0.31415926535897931e1
```

```
SingleValue eq INF
```

```
DecimalValue eq 34.95
```

```
StringValue eq 'Say Hello,then go'

DateValue eq 2012-12-03

DateTimeOffsetValue eq 2012-12-03T07:16:23Z

DurationValue eq duration'P12DT23H59M59.999999999999S'

DurationValue eq 'P12DT23H59M59.999999999999S'

TimeOfDayValue eq 07:59:59.999

GuidValue eq 01234567-89ab-cdef-0123-456789abcdef

Int64Value eq 0

ColorEnumValue eq Sales.Pattern'Yellow'

ColorEnumValue eq 'Yellow'

geo.distance(Location,geography'SRID=0;Point(142.1 64.1)')
```

Duration literals in OData 4.0 required prefixing with "**`duration`**". Enumeration literals in OData 4.0 required prefixing with the qualified type name of the enumeration.

In OData 4.01, services MUST support duration and enumeration literals with or without the type prefix. OData clients that want to operate across OData 4.0 and OData 4.01 services should always include the prefix for duration and enumeration types.

5.1.1.14.2 Structured and Collection Literals

Complex literals and collection literals in URLs are represented as JSON objects and arrays according to the **`arrayOrObject`** rule in **[OData-ABNF]**. Such literals MUST NOT appear in the path portion of the URL but can be passed to bound functions and function imports in path segments by using parameter aliases.

Object member values and array items can be expressions, including other objects and arrays, arithmetic expressions, property names, and of course primitive values.

Note that the special characters `{`, `}`, `[`, `]`, and `"` MUST be percent-encoded in URLs although some browsers will accept and pass them on unencoded.

Example 110: collection of string literals

```
http://host/service/ProductsByColors(colors=@c)?@c=["red","green"]
```

Example 111: check whether a pair of properties has one of several possible pair values

```
$filter=[FirstName,LastName] in [["John","Doe"],["Jane","Smith"]]
```

Entities are represented as structured literals as described in **[OData-JSON, section 6]**. Non-transient entities can alternatively be represented through their resource path.

Example 112: determine the price of an adhoc-defined product

```
http://host/service/Price(Product=@p)?@p={"Color":"red"}
```

5.1.1.14.3 null

The **null** literal can be used to compare a value to null, or to pass a null value to a function.

5.1.1.14.4 $it

The **$it** literal can be used in expressions to refer to the current instance of the resource identified by the resource path. For a collection-valued resource the current instance is the instance currently being evaluated by the system query option. For a single-valued resource it is the resource itself.

It allows comparing properties of related entities to properties of the current instance in expressions within lambda operators or in **$filter** expressions nested within **$expand** or **$select**.

It also can be used in **$filter** and **$orderby** expressions on collections of primitive types to refer to the current primitive instance, but using the **$this** literal is preferred as **$this** can also be used in **$filter** and **$orderby** expressions nested within **$select**.

Note: property names and property paths in **$filter** expressions nested within **$expand** are evaluated in the context of the declared type of the expanded navigation property, so property names and property paths for the current instance of the collection identified by the resource path MUST be prefixed with **$it/**.

The **$it** literal can also be used as a path prefix to invoke a bound function overload on the current instance within an expression. Function names without a path prefix refer to an unbound function overload.

Example 113: email addresses ending with **.com** *assuming* **EmailAddresses** *is a collection of strings*

```
http://host/service/Customers(1)/EmailAddresses?$filter=endswith($it,'.com')
```

Example 114: customers along with their orders that shipped to the same city as the customer's address. The nested filter expression is evaluated in the context of Orders; **$it** *allows referring to values in the outer context of Customers. Note: the nested filter condition could equivalently be expressed as* **$it/Address/City eq $this/ShipTo/City**.

```
http://host/service/Customers?$expand=Orders($filter=$it/Address/City eq ShipTo/City)
```

Example 115: products with at least 10 positive reviews. **Model.PositiveReviews** *is a function bound to* **Model.Product** *returning a collection of reviews.*

```
http://host/service/Products?$filter=$it/Model.PositiveReviews()/$count ge 10
```

5.1.1.14.5 $root

The **$root** literal can be used in expressions to refer to resources of the same service.

Example 116: all employees with the same last name as employee **A1235**

```
http://host/service/Employees?$filter=LastName eq $root/Employees('A1245')/LastName
```

Example 117: products ordered by a set of customers, where the set of customers is passed as a JSON array containing the resource paths from **$root** *to each customer*

```
http://host/service/ProductsOrderedBy(Customers=@c)
  ?@c=[$root/Customers('ALFKI'),$root/Customers('BLAUS')]
```

Example 118: function call returning the average rating of a given employee by their peers (employees in department D1)

```
http://host/service/Employees('A1245')/self.AvgRating(RatedBy=@peers)
  ?@peers=$root/Employees/$filter(Department eq 'D1')
```

5.1.1.14.6 $this

The **$this** literal can be used in **$filter** and **$orderby** expressions nested within **$expand** and **$select** for collection-valued properties and navigation properties. It refers to the current instance of the collection.

Example 119: select only email addresses ending with **.com**

```
http://host/service/Customers?$select=EmailAddresses($filter=endswith($this,'.com'))
```

5.1.1.15 Path Expressions

Properties and navigation properties of the structured type on which a common expression is evaluated can be used as operands or function parameters, as shown in the preceding examples.

Properties of complex properties can be used via the same syntax as in resource paths, i.e. by specifying the name of a complex property, followed by a forward slash (**/**) and the name of a property of the complex property, and so on,

Properties and navigation properties of entities related with a target cardinality 0..1 or 1 can be used by specifying the navigation property, followed by a forward slash (**/**) and the name of a property of the related entity, and so on.

If a complex property is **null**, or no entity is related (in case of target cardinality 0..1), its value, and the values of its components, are treated as **null**.

Example 120: similar behavior whether **HeadquarterAddress** *is a nullable complex type or a nullable navigation property*

```
Companies(1)/HeadquarterAddress/Street
```

To access properties of derived types, the property name MUST be prefixed with the qualified name of the derived type on which the property is defined, followed by a forward slash (**/**), see addressing derived types. If the current instance is not of the specified derived type, the path expression returns **null**.

If the property or navigation property is not defined for the type of the resource and that type supports dynamic properties or navigation properties, then the property or navigation property is treated as **null** for all instances on which it has no value.

If the property or navigation property is not defined for the type of the resource and that type does not support dynamic properties or navigation properties, then the request may be considered malformed.

5.1.1.16 Annotation Values in Expressions

Services MAY support the use of annotation values as operands or function parameters, and they MAY advertise this by annotating the entity container with term **Capabilities.AnnotationValuesInQuerySupported**, see **[OData-VocCap]**.

Annotation values are referenced by the annotation name which consists of an at sign (**@**) followed by the qualified term name, optionally followed by a percent-encoded hash (**%23**) and an annotation qualifier. The annotation name can be prefixed with a path expression leading to the annotated resource or property.

If an annotation is not applied to the resource or property, then its value, and the values of its components, are treated as **null**.

Example 121: Return Products that have prices in Euro

```
http://host/service/Products?$filter=Price/@Measures.Currency eq 'EUR'
```

Example 122: Return Employees that have any error messages in the **Core.Messages** *annotation*

```
http://host/service/Employees?$filter=@Core.Messages/any(m:m/severity eq 'error')
```

Services MAY additionally support the use of the unqualified term name by defining one or more default namespaces through the **Core.DefaultNamespace** annotation term defined in **[OData-VocCore]**. For more information on default namespaces, see **[OData-Protocol, section 4.4]**. This short notation however uses the same name pattern as parameter aliases. If a query option is specified as a parameter alias, then any occurrence of the parameter alias name in an expression MUST evaluate to the parameter alias value and MUST NOT evaluate to the annotation value of an identical unqualified term name.

5.1.1.17 Operator Precedence

OData services MUST use the following operator precedence for supported operators when evaluating **$filter** and **$orderby** 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	ABNF Expression
Grouping	**( )**	Precedence grouping	**parenExpr** **boolParenExpr**
Primary	**/**	Navigation	**firstMemberExpr** **memberExpr**
	has	Enumeration Flags	**hasExpr**
	in	Is a member of	**inExpr**
	xxx( )	Method Call	**methodCallExpr** **boolMethodCallExpr** **functionExpr**
Unary	**-**	Negation	**negateExpr**
	not	Logical Negation	**notExpr**
	cast( )	Type Casting	**castExpr**
Multiplicative	**mul**	Multiplication	**mulExpr**
	div	Division	**divExpr**
	divby	Decimal Division	**divbyExpr**
	mod	Modulo	**modExpr**
Additive	**add**	Addition	**addExpr**
	sub	Subtraction	**subExpr**
Relational	**gt**	Greater Than	**gtExpr**
	ge	Greater than or Equal	**geExpr**
	lt	Less Than	**ltExpr**

Group	Operator	Description	ABNF Expression
	le	Less than or Equal	**leExpr**
	isof	Type Testing	**isofExpr**
Equality	**eq**	Equal	**eqExpr**
	ne	Not Equal	**neExpr**
Conditional AND	**and**	Logical And	**andExpr**
Conditional OR	**or**	Logical Or	**orExpr**

5.1.1.18 Numeric Promotion

Services SHOULD NOT require explicit cast operations between numeric types used in comparison expressions. Wherever possible, such comparisons should be performed using underlying types of sufficient size.

Services MAY support numeric promotion for arithmetic operations or when comparing two operands of comparable types by applying the following rules, in order:

- If either operand is **Edm.Double**, the other operand is converted to type **Edm.Double**.
- Otherwise, if either operand is **Edm.Single**, the other operand is converted to type **Edm.Single**.
- Otherwise, if either operand is of type **Edm.Decimal**, the other operand is converted to **Edm.Decimal**.
- Otherwise, if either operand is **Edm.Int64**, the other operand is converted to type **Edm.Int64**.
- Otherwise, if either operand is **Edm.Int32**, the other operand is converted to type **Edm.Int32**.
- Otherwise, if either operand is **Edm.Int16**, the other operand is converted to type **Edm.Int16**.

Each of these promotions uses the same semantics as a **castExpression** to promote an operand to the target type.

OData does not define an implicit conversion between string and numeric types.

5.1.2 System Query Option `$filter`

The **$filter** system query option allows clients to filter a collection of resources that are addressed by a request URL. The expression specified with **$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.

The [OData-ABNF] **filter** syntax rule defines the formal grammar of the **$filter** query option.

5.1.3 System Query Option `$expand`

The **$expand** system query option specifies the related resources or media streams to be included in line with retrieved resources.

The [OData-ABNF] **expand** syntax rule defines the formal grammar of the **$expand** query option.

The value of **$expand** is a comma-separated list of expand items. Each expand item is evaluated relative to the retrieved resource being expanded. An expand item is either a path or one of the symbols ***** or **$value**.

A path consists of segments separated by a forward slash (**/**). Segments are either names of single- or collection-valued complex properties, instance annotations, or type-cast segments consisting of the qualified name of a structured type that is derived from the type identified by the preceding path segment to reach properties defined on the derived type.

A path can end with

- the name of a stream property to include that stream property,
- a star (*****) to expand all navigation properties of the identified instances of a structured type, optionally followed by **/$ref** to expand only entity references, or
- a navigation property to expand the related entity or entities, optionally followed by a type-cast segment to expand only related entities of that derived type or one of its sub-types, optionally followed by **/$ref** to expand only entity references, or **/$count**, optionally with **$filter** and/or **$search** expand options, to return only the count of matching entities.
- an entity-valued instance annotation to expand the related entity or entities, optionally followed by a type-cast segment to expand only related entities of that derived type or one of its sub-types.

A path MUST NOT appear in more than one expand item.

If a structured type traversed by the path supports neither dynamic properties nor instance annotations, then a corresponding property segment MUST specify a declared property of that structured type. Otherwise, if a traversed type does support dynamic navigation properties or instance annotations and the corresponding property segment does not specify a declared property, then the expanded property appears only for those instances on which it has a value.

Example 123: expand a navigation property of an entity type

```
http://host/service/Products?$expand=Category
```

Example 124: expand a navigation property of a complex type

```
http://host/service/Customers?$expand=Addresses/Country
```

Example 125: all categories and for each category the number of all related products

```
http://host/service/Categories?$expand=Products/$count
```

Example 126: all categories and for each category the number of all related blue products

```
http://host/service/Categories?$expand=Products/$count($search=blue)
```

To retrieve entity references instead of the related entities, append **/$ref** to the navigation property name or type-cast segment following a navigation property name. The Expand Options **$filter**, **$search**, **$skip**, and **$top** can be used to limit the collection of expanded entity references, and **$count** can be used to include the count of expanded entity references.

Example 127: all categories and for each category the references of all related products

```
http://host/service/Categories?$expand=Products/$ref
```

Example 128: all categories and for each category the references of all related products of the derived type **Sales.PremierProduct**

```
http://host/service/Categories?$expand=Products/Sales.PremierProduct/$ref
```

Example 129: all categories and for each category the references of all related premier products with a current promotion equal to **null**

```
http://host/service/Categories
  ?$expand=Products/Sales.PremierProduct/$ref($filter=CurrentPromotion eq null)
```

It is also possible to expand all declared and dynamic navigation properties using a star (**`*`**). To retrieve references to all related entities use **`*/$ref`**, and to expand all related entities with a certain distance use the star operator with the **`$levels`** Expand Option. The star operator can be combined with explicitly named navigation properties, which take precedence over the star operator.

The star operator does not implicitly include stream properties.

Example 130: expand **`Supplier`** *and include references for all other related entities*

```
http://host/service/Categories?$expand=*/$ref,Supplier
```

Specifying a stream property includes the media stream inline according to the specified format.

Example 131: include Employee's **`Photo`** *stream property along with other properties of the customer*

```
http://host/service/Employees?$expand=Photo
```

Specifying **`$value`** for a media entity includes the media entity's stream value inline according to the specified format.

Example 132: Include the Product's media stream along with other properties of the product

```
http://host/service/Products?$expand=$value
```

5.1.3.1 Expand Options

Query options can be applied to an expanded navigation property by appending a semicolon-separated list of query options, enclosed in parentheses, to the navigation property name. The system query option, irrespective of casing or whether or not it is prefixed with a **`$`**, MUST NOT be specified more than once in the list. Allowed system query options are **`$compute`**, **`$select`**, **`$expand`**, and **`$levels`** for all navigation properties, plus **`$filter`**, **`$orderby`**, **`$skip`**, **`$top`**, **`$count`**, and **`$search`** for collection-valued navigation properties.

Example 133: all categories and for each category all related products with a discontinued date equal to **`null`**

```
http://host/service/Categories?$expand=Products($filter=DiscontinuedDate eq null)
```

Cyclic navigation properties (whose target type is identical or can be cast to its source type) can be recursively expanded using the special **`$levels`** expand option. The value of the **`$levels`** expand option is either a positive integer to specify the number of levels to expand, or the literal string **`max`** to specify the maximum expansion level supported by that service. A **`$levels`** option with a value of 1 specifies a single expand with no recursion.

Example 134: all employees with their manager, manager's manager, and manager's manager's manager

```
http://host/service/Employees?$expand=ReportsTo($levels=3)
```

Example 135: expand all related entities and their related entities

```
http://host/service/Categories?$expand=*($levels=2)
```

5.1.4 System Query Option `$select`

The **`$select`** system query option allows clients to request a specific set of properties for each entity or complex type.

The **$select** query option is often used in conjunction with the **$expand** system query option, to define the extent of the resource graph to return (**$expand**) and then specify a subset of properties for each resource in the graph (**$select**). Expanded navigation properties MUST be returned, even if they are not specified in **$select**.

The [OData-ABNF] **select** syntax rule defines the formal grammar of the **$select** query option.

The value of **$select** is a comma-separated list of select items. Each select item is one of the following:

- a path, optionally followed by a count segment or select options
- a star (*****), to include all declared or dynamic properties of the type, or
- a qualified schema name followed by a dot (**.**) followed by a star (*****) to request all applicable actions or functions from that schema

A path consists of segments separated by a forward slash (**/**). Segments are either names of single- or collection-valued complex properties, instance annotations, or type-cast segments consisting of the qualified name of a structured type that is derived from the type identified by the preceding path segment to reach properties defined on the derived type.

A path can end with

- the name of a property or non-entity-valued instance annotation of the identified instance of a structured type,
- the qualified name of a bound action,
- the qualified name of a bound function to include all matching overloads, or
- the qualified name of a bound function followed by parentheses containing the comma-separated lists of non-binding parameters identifying a single overload.

The **$select** system query option is interpreted relative to the entity type or complex type of the resources identified by the resource path section of the URL. Each select item in the **$select** clause indicates that the response MUST include the declared or dynamic properties, actions and functions identified by that select item. If a select item is a path expression traversing an entity or complex property that is **null** on an instance, then the null-valued entity or complex property is included and represented as **null**. The simplest form of a select item explicitly requests a property defined on the entity type of the resources identified by the resource path section of the URL.

Example 136: rating and release date of all products

```
http://host/service/Products?$select=Rating,ReleaseDate
```

It is also possible to request all declared and dynamic structural properties using a star (*****).

Example 137: all structural properties of all products

```
http://host/service/Products?$select=*
```

If the select item is not defined for the type of the resource, and that type supports dynamic properties or instance annotations, then the property is treated as **null** for all instances on which it is not defined.

If the select item is not defined for the type of the resource, and that type does not support dynamic properties or instance annotations, then the request is considered malformed.

If the select item is an instance annotation of type entity or collection of entities, then the request is considered malformed. Entity-valued annotations can be included using **$expand**.

If the select item is a navigation property, then the corresponding navigation link is represented in the response. If the navigation property also appears in an **$expand** query option, then it is additionally represented as inline content. This inline content can itself be restricted with a nested **$select** query option, see section 5.1.2.

Example 138: name and description of all products, plus name of expanded category

```
http://host/service/Products?$select=Name,Description
  &$expand=Category($select=Name)
```

The select item MUST be prefixed with a qualified structured type name in order to select a property defined on a type derived from the type of the resource segment.

A select item that is a complex type or collection of complex type can be followed by a forward slash, an optional type-cast segment, and the name of a property of the complex type (and so on for nested complex types).

Example 139: the **`AccountRepresentative`** *property of any supplier that is of the derived type* **`Namespace.PreferredSupplier`**, *together with the* **`Street`** *property of the complex property* **`Address`**, *and the Location property of the derived complex type* **`Namespace.AddressWithLocation`**

```
http://host/service/Suppliers
  ?$select=Namespace.PreferredSupplier/AccountRepresentative,
           Address/Street,Address/Namespace.AddressWithLocation/Location
```

If the path ends in a collection of primitive or complex values, then the count segment (**`/$count`**), optionally followed by the Select Options **`$filter`** and/or **`$search`**, can be appended to the path in order to return only the count of the matching items.

Example 140: for each **`Customer`**, *return the* **`ID`** *and the count of* **`Addresses`**

```
http://host/service/Customers?$select=ID,Addresses/$count
```

Example 141: for each **`Customer`**, *return the* **`ID`** *and the count of* **`Addresses`** *whose* **`City`** *starts with 'H'*

```
http://host/service/Customers?$select=ID,Addresses/$count($filter=startswith(City,'H'))
```

Any structural property, non-expanded navigation property, or operation not requested as a select item (explicitly or via a star) SHOULD be omitted from the response.

Annotations requested in **`$select`** MUST be included in the response; **`$select`** overrules the **`include-annotations`** preference (see **[OData-Protocol, section 8.2.8.4]**) for the explicitly requested annotations. Additional annotations matching the preference can be included even if not requested via **`$select`**. The **`Preference-Applied`** response header only reflects the set of annotations included due to the **`include-annotations`** preference and not those only included due to **`$select`**.

If any select item (including a star) is specified, actions and functions SHOULD be omitted unless explicitly requested.

If an action or function is requested as a select item, either explicitly by using its qualified name, or implicitly by requesting all operations in a schema, then the service includes information about how to invoke that operation for each entity identified by the last path segment in the request URL for which the operation can be bound.

If an action or function is requested in a select item using its qualified name and that operation cannot be bound to the entities requested, the service MUST ignore the select item.

Example 142: the **`ID`** *property, the* **`ActionName`** *action defined in* **`Model`** *and all actions and functions defined in the* **`Model2`** *for each product if those actions and functions can be bound to that product*

```
http://host/service/Products?$select=ID,Model.ActionName,Model2.*
```

When multiple select item exist in a **`$select`** clause, then the total set of properties, open properties, navigation properties, actions and functions to be returned is equal to the union of the set of those identified by each select item.

5.1.4.1 Select Options

Query options can be applied to a select item that is a path to a single complex value or a collection of primitive or complex values by appending a semicolon-separated list of query options, enclosed in parentheses, to the select item. The allowed sytem query options are **$compute** and **$select** for all complex-typed properties, plus **$filter**, **$orderby**, **$skip**, **$top**, **$count**, and **$search** for collection-valued properties. The same property MUST NOT have select options specified in more than one place in a request.

If the select item is a complex type, or collection of complex types, then it can include a nested select.

Example 143: return the City from the Address complex type

```
http://host/service/Customers?$select=Address($select=City)
```

Example 144: select up to five addresses whose **City** *starts with an* **H**, *sorted, and with the* **Country** *expanded*

```
http://host/service/Customers
  ?$select=Addresses($filter=startswith(City,'H');$top=5;
                     $orderby=Country/Name,City,Street)
  &$expand=Addresses/Country
```

5.1.5 System Query Option $orderby

The **$orderby** system query option allows clients to request resources in a particular order.

The semantics of **$orderby** are covered in **[OData-Protocol, section 11.2.6.2]**.

The **[OData-ABNF]** **orderby** syntax rule defines the formal grammar of the **$orderby** query option.

5.1.6 System Query Options $top and $skip

The **$top** system query option requests the number of items in the queried collection to be included in the result. The **$skip** query option requests the number of items in the queried collection that are to be skipped and not included in the result. A client can request a particular page of items by combining **$top** and **$skip**.

The semantics of **$top** and **$skip** are covered in **[OData-Protocol, section 11.2.6.3]** and **[OData-Protocol, section 11.2.6.4]**. The **[OData-ABNF]** **top** and **skip** syntax rules define the formal grammar of the **$top** and **$skip** query options respectively.

5.1.7 System Query Option $count

The **$count** system query option allows clients to request a count of the matching resources included with the resources in the response. The **$count** query option has a Boolean value of **true** or **false**.

The semantics of **$count** is covered in **[OData-Protocol, section 11.2.6.5]**.

5.1.8 System Query Option $search

The **$search** system query option allows clients to request items within a collection matching a free-text search expression.

The **$search** query option can be applied to a URL representing a collection of entity, complex, or primitive typed instances, to return all matching items within the collection. Applying the **$search** query option to the **$all** resource requests all matching entities in the service.

If both **$search** and **$filter** are applied to the same request, the results include only those items that match both criteria.

The [OData-ABNF] **search** syntax rule defines the formal grammar of the **$search** query option.

Example 145: all products that are blue or green. It is up to the service to decide what makes a product blue or green.

```
http://host/service/Products?$search=blue OR green
```

Clients should be aware that services MAY implement search based on a different syntax provided they advertise this with the annotation **SearchRestrictions/SearchSyntax** defined in [OData-VocCap]. Services MAY treat keywords defined in the standard [OData-ABNF] **$search** syntax as terms to be matched if they are listed in **SearchRestrictions/UnsupportedExpressions**.

5.1.8.1 Search Expressions

Search expressions are used within the **$search** system query option to request entities matching the specified expression. Leading and trailing spaces are not considered part of the search expression.

Terms can be any single word to be matched within the expression.

Terms enclosed in double-quotes comprise a *phrase*.

Each individual term or phrase comprises a Boolean expression that returns **true** if the term or phrase is matched, otherwise **false**. The semantics of what is considered a match is dependent upon the service.

Expressions enclosed in parenthesis comprise a *group expression*.

The search expression can contain any number of terms, phrases, or group expressions, along with the case-sensitive keywords **NOT**, **AND**, and **OR**, evaluated in that order.

Expressions prefaced with **NOT** evaluate to **true** if the expression is not matched, otherwise **false**.

Two expressions not enclosed in quotes and separated by a space are equivalent to the same two expressions separated by the **AND** keyword. Such expressions evaluate to **true** if both expressions evaluate to **true**, otherwise **false**.

Expressions separated by an **OR** evaluate to **true** if either of the expressions evaluate to **true**, otherwise **false**.

To support type-ahead use cases, incomplete search expressions can be sent as OData string literals enclosed in single-quotes, and single-quotes within the search expression doubled. Such an expression can also be used to search for double quotes: **?$search='"'**.

The [OData-ABNF] **searchExpr** syntax rule defines the formal grammar of the search expression.

5.1.9 System Query Option $format

The **$format** system query option allows clients to request a response in a particular format and is useful for clients without access to request headers for standard content-type negotiation. Where present **$format** takes precedence over standard content-type negotiation.

The semantics of **$format** is covered in [OData-Protocol, section 11.2.11].

The [OData-ABNF] **format** syntax rule defines the formal grammar of the **$format** query option.

5.1.10 System Query Option $compute

The **$compute** system query option allows clients to define computed properties that can be used in a **$select** or within a **$filter** or **$orderby** expression.

The **$compute** system query option is interpreted relative to the entity type or complex type of the resources identified by the resource path section of the URL.

The value of **$compute** is a comma-separated list of compute instructions, each consisting of a common expression followed by the keyword **as**, followed by the name for the computed dynamic property. This name MUST differ from the names of declared or dynamic properties of the identified resources. Services MAY support compute instructions that address dynamic properties added by other compute instructions within the same **$compute** system query option, provided that the service can determine an evaluation sequence.

The **[OData-ABNF]** **compute** syntax rule defines the formal grammar of the **$compute** query option.

Computed properties SHOULD be included as dynamic properties in the result and MUST be included if **$select** is specified with the computed property name, or star (*****).

Example 146: compute total price for order items

```
http://host/service/Orders(10)/Items
  ?$select=Product/Description,Total
  &$filter=Total gt 100
  &$orderby=Total
  &$compute=Product/Price mul Quantity as Total
```

5.1.11 System Query Option $index

The **$index** system query option allows clients to do a positional insert into a collection annotated with the **Core.PositionalInsert** term (see **[OData-VocCore]**). The value of the **$index** system query option is the zero-based ordinal position where the item is to be inserted. The ordinal of items within the collection greater than or equal to the inserted position are increased by one. A negative ordinal indexes from the end of the collection, with -1 representing an insert at the end of the collection.

The **[OData-ABNF]** **index** syntax rule defines the formal grammar of the **$index** query option.

5.1.12 System Query Option $schemaversion

The **$schemaversion** system query option allows clients to specify the version of the schema against which the request is made. The semantics of **$schemaversion** is covered in **[OData-Protocol, section 11.2.12]**.

The **[OData-ABNF]** **schemaversion** syntax rule defines the formal grammar of the **$schemaversion** query option

5.2 Custom Query Options

Custom query options provide an extensible mechanism for service-specific information to be placed in a URL query string. A custom query option is any query option of the form shown by the rule **customQueryOption** in **[OData-ABNF]**.

Custom query options MUST NOT begin with a **$** or **@** character.

Example 147: service-specific custom query option **debug-mode**

```
http://host/service/Products?debug-mode=true
```

5.3 Parameter Aliases

Parameter aliases can be used in place of literal values in entity keys, function parameters, or within a **$filter** or **$orderby** expression.

Parameter aliases MUST start with an **@** character, see rule **parameterAlias** in **[OData-ABNF]**.

The semantics of parameter aliases are covered in **[OData-Protocol, section 11.2.6.1.3]**. The **[OData-ABNF]** rule **`aliasAndValue`** defines the formal grammar for passing parameter alias values as query options.

Example 148:

```
http://host/service/Movies?$filter=contains(@word,Title)&@word='Black'
```

Example 149:

```
http://host/service/Movies?$filter=Title eq @title&@title='Wizard of Oz'
```

Example 150: JSON array of strings as parameter alias value — note that `[`, `]`, and `"` need to be percent-encoded in real URLs, the clear-text representation used here is just for readability

```
http://host/service/Products/Model.WithIngredients(Ingredients=@i)
  ?@i=["Carrots","Ginger","Oranges"]
```

6 Conformance

The conformance requirements for OData clients and services are described in **[OData-Protocol, section 12]**.

Appendix A. References

This appendix contains the normative and informative references that are used in this document.

While any hyperlinks included in this appendix were valid at the time of publication, OASIS cannot guarantee their long-term validity.

A.1 Normative References

The following documents are referenced in such a way that some or all of their content constitutes requirements of this document.

[OData-ABNF]

ABNF components: OData ABNF Construction Rules Version 4.02 and OData ABNF Test Cases.
See link in "Related work" section on cover page.

[OData-CSDL]

OData Common Schema Definition Language (CSDL) JSON Representation Version 4.02.
See link in "Related work" section on cover page.

OData Common Schema Definition Language (CSDL) XML Representation Version 4.02.
See link in "Related work" section on cover page.

[OData-JSON]

OData JSON Format Version 4.02.
See link in "Related work" section on cover page.

[OData-Protocol]

OData Version 4.02. Part 1: Protocol.
See link in "Related work" section on cover page.

[OData-VocCap]

OData Vocabularies Version 4.0: Capabilities Vocabulary.
See link in "Related work" section on cover page.

[OData-VocCore]

OData Vocabularies Version 4.0: Core Vocabulary.
See link in "Related work" section on cover page.

[RFC2119]

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997. https://www.rfc-editor.org/info/rfc2119.

[RFC3986]

Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005. https://www.rfc-editor.org/info/rfc3986.

[RFC8174]

Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017. https://www.rfc-editor.org/info/rfc8174.

[URL]

URL Living Standard.
https://url.spec.whatwg.org/.

[XML-Schema-2]

W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. D. Peterson, S. Gao, C. M. Sperberg-McQueen, H. S. Thompson, P. V. Biron, A. Malhotra, Editors, W3C Recommendation, 5 April 2012. http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/. Latest version available at http://www.w3.org/TR/xmlschema11-2/.

A.2 Informative References

[ECMAScript]

ECMAScript 2023 Language Specification, 14th Edition, June 2023. Standard ECMA-262. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

[Well-Known Text]

OpenGIS Implementation Specification for Geographic information – Simple feature access – Part 1: Common architecture, May 2011. Open Geospatial Consortium. https://www.ogc.org/standard/sfa/.

Appendix B. Acknowledgments

B.1 Participants

OData TC Members:

First Name	Last Name	Company
George	Ericson	Dell
Hubert	Heijkers	IBM
Ling	Jin	IBM
Stefan	Hagen	Individual
Michael	Pizzo	Microsoft
Christof	Sprenger	Microsoft
Ralf	Handl	SAP SE
Gerald	Krause	SAP SE
Heiko	Theißen	SAP SE

Appendix C. Revision History

Revision	Date	Editor	Changes Made
Committee Specification Draft 01	2024-02-28	Michael Pizzo Ralf Handl Heiko Theißen	Import material from OData Version 4.01 Part 2: URL Conventions Changes listed in section 1.1

Appendix D. Notices

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the “OASIS IPR Policy”). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an “AS IS” basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards Final Deliverable documents (Committee Specification, Candidate OASIS Standard, OASIS Standard, or Approved Errata).

[OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Standards Final Deliverable, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this deliverable.]

[OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this OASIS Standards Final Deliverable by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this OASIS Standards Final Deliverable. OASIS may include such claims on its website, but disclaims any obligation to do so.]

[OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this OASIS Standards Final Deliverable or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS’ procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Standards Final Deliverable, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.]

The name “OASIS” is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see https://www.oasis-open.org/policies-guidelines/trademark/ for above guidance.