OASIS ⦿

OData Version 4.0 Part 2: URL Conventions

Candidate OASIS Standard 01

19 November 2013

Specification URIs
This version:
http://docs.oasis-open.org/odata/odata/v4.0/cos01/part2-url-conventions/odata-v4.0-cos01-part2-url-conventions.doc (Authoritative)
http://docs.oasis-open.org/odata/odata/v4.0/cos01/part2-url-conventions/odata-v4.0-cos01-part2-url-conventions.html
http://docs.oasis-open.org/odata/odata/v4.0/cos01/part2-url-conventions/odata-v4.0-cos01-part2-url-conventions.pdf

Previous version:
http://docs.oasis-open.org/odata/odata/v4.0/cs01/part2-url-conventions/odata-v4.0-cs01-part2-url-conventions.doc (Authoritative)
http://docs.oasis-open.org/odata/odata/v4.0/cs01/part2-url-conventions/odata-v4.0-cs01-part2-url-conventions.html
http://docs.oasis-open.org/odata/odata/v4.0/cs01/part2-url-conventions/odata-v4.0-cs01-part2-url-conventions.pdf

Latest version:
http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part2-url-conventions.doc (Authoritative)
http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part2-url-conventions.html
http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part2-url-conventions.pdf

Technical Committee:
OASIS Open Data Protocol (OData) TC

Chairs:
Barbara Hartel (barbara.hartel@sap.com), SAP AG
Ram Jeyaraman (Ram.Jeyaraman@microsoft.com), Microsoft

Editors:
Michael Pizzo (mikep@microsoft.com), Microsoft
Ralf Handl (ralf.handl@sap.com), SAP AG
Martin Zurmuehl (martin.zurmuehl@sap.com), SAP AG

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

* *OData Version 4.0 Part 1: Protocol*. http://docs.oasis-open.org/odata/odata/v4.0/cos01/part1-protocol/odata-v4.0-cos01-part1-protocol.html
* *OData Version 4.0 Part 2: URL Conventions* (this document). http://docs.oasis-open.org/odata/odata/v4.0/cos01/part2-url-conventions/odata-v4.0-cos01-part2-url-conventions.html
* *OData Version 4.0 Part 3: Common Schema Definition Language (CSDL)*. http://docs.oasis-open.org/odata/odata/v4.0/cos01/part3-csdl/odata-v4.0-cos01-part3-csdl.html
* ABNF components: *OData ABNF Construction Rules Version 4.0* and *OData ABNF Test Cases*. http://docs.oasis-open.org/odata/odata/v4.0/cos01/abnf/

- Vocabulary components: *OData Core Vocabulary, OData Measures Vocabulary* and *OData Capabilities Vocabulary*. http://docs.oasis-open.org/odata/odata/v4.0/cos01/vocabularies/
- XML schemas: OData EDMX XML Schema and OData EDM XML Schema. http://docs.oasis-open.org/odata/odata/v4.0/cos01/schemas/
- OData Metadata Service Entity Model: http://docs.oasis-open.org/odata/odata/v4.0/cos01/models/MetadataService.edmx

Related work:

This specification is related to:

- *OData Atom Format Version 4.0*. Latest version. http://docs.oasis-open.org/odata/odata-atom-format/v4.0/odata-atom-format-v4.0.html.
- *OData JSON Format Version 4.0*. Latest version. http://docs.oasis-open.org/odata/odata-json-format/v4.0/odata-json-format-v4.0.html.

Declared XML namespaces:

- http://docs.oasis-open.org/odata/ns/edmx
- http://docs.oasis-open.org/odata/ns/edm

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 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.

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 version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at http://www.oasis-open.org/committees/odata/.

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 Technical Committee web page (http://www.oasis-open.org/committees/odata/ipr.php).

Citation format:

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

[OData-Part2]

OData Version 4.0 Part 2: URL Conventions. 19 November 2013. Candidate OASIS Standard 01. http://docs.oasis-open.org/odata/odata/v4.0/cos01/part2-url-conventions/odata-v4.0-cos01-part2-url-conventions.html.

Notices

Copyright © OASIS Open 2013. All Rights Reserved.

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.

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 Committee Specification or OASIS Standard, 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 specification.

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 specification 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 specification. 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 document 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 Committee Specification or OASIS Standard, 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 http://www.oasis-open.org/policies-guidelines/trademark for above guidance.

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-Atom]** and **[OData-JSON]** documents specify 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 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in **[RFC2119]**.

1.2 Normative References

[OData-ABNF]	*OData ABNF Construction Rules Version 4.0*. See the link in "Additional artifacts" section on cover page.
[OData-Atom]	*OData ATOM Format Version 4.0*. See link in "Related work" section on cover page.
[OData-CSDL]	*OData Version 4.0 Part 3: Common Schema Definition Language (CSDL)*. See link in "Additional artifacts" section on cover page.
[OData-JSON]	*OData JSON Format Version 4.0.* See link in "Related work" section on cover page.
[OData-Protocol]	*OData Version 4.0 Part 1: Protocol*. See link in "Additional artifacts" section on cover page.
[OData-VocCore]	*OData Core Vocabulary.* See link in "Additional artifacts" section on cover page.
[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997. http://www.ietf.org/rfc/rfc2119.txt.
[RFC3986]	Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005. http://www.ietf.org/rfc/rfc3986.txt.
[RFC5023]	Gregorio, J., Ed., and B. de hOra, Ed., "The Atom Publishing Protocol.", RFC 5023, October 2007. http://tools.ietf.org/html/rfc5023.

1.3 Typographical Conventions

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

```
Normative source code uses this paragraph style.
```

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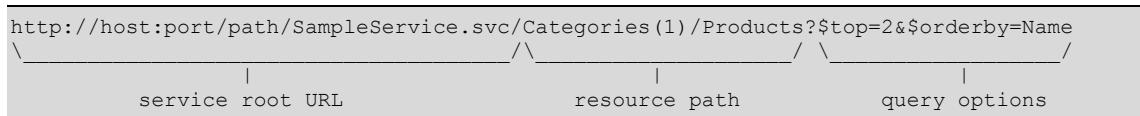
Mandated and suggested content of these three significant URL components used by an OData service are covered in sequence in the three following chapters.

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 at first ":", then first "?", and then first "#"
- Split undecoded hier-part into authority and path
- Split undecoded path into path segments at "/"

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

- Split undecoded query at "&" into query options, and each query option at the first "=" into query option name and query option value
- Percent-decode path segments, query option names, and query option values
- Interpret path segments, query option names, and query option values according to OData rules

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] and [ODATA-Atom].

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 `Content-Location` (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 services MAY expose their entity model as a service, according to **[OData-CSDL]**, by appending a trailing slash (`/`) to the metadata document URL.

Example 6: Metadata service root URL

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

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]
             / singleton                      [singleNavigation]
             / actionImportCall
             / entityColFunctionImportCall    [ collectionNavigation ]
             / entityFunctionImportCall       [ singleNavigation ]
             / complexColFunctionImportCall   [ collectionPath ]
             / complexFunctionImportCall      [ complexPath ]
             / primitiveColFunctionImportCall [ collectionPath ]
             / primitiveFunctionImportCall    [ singlePath ]
             / crossjoin
             / '$all'
```

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 that returns a collection of entities (see rule: `entityColFunctionCall`)

   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 that returns a collection of entities (see rule: `actionCall`)

Likewise there are many ways to address a single entity.

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

- Invoking a function that returns a single entity (see rule: `entityFunctionCall`)

- Invoking an action that returns a single entity (see rule: `actionCall`)

- 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)
   ```

- 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 13:

```
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 14:

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

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

  Example 15:

```
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 16:

```
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 17:

```
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 18:

```
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.

Example 19: Non-canonical URL

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

Example 20: 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 navigation property that specifies `ContainsTarget="true"`, see **[OData-CSDL]**) the canonical URL is the canonical URL of the containing entity followed by:

* A 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.

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 21: full key predicate of related entity

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

Example 22: 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` 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 service root URL.

Example 23: request the entity representation for an entity-id

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

The semantics of `$entity` are covered in the **[OData-Protocol]** document.

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 24: URL addressing the references between `Categories(1)` and `Products`

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

Resource paths addressing a single entity reference can be used in `DELETE` requests to unrelated two entities. Resource paths addressing collection of references can be used in `DELETE` requests if they are followed by the system query option `$id` identifying one of the entity references in the collection. The entity-id specified by `$id` may be expressed absolute or relative to the request URL. For details see **[OData-Protocol]**.

Example 25: two ways of unrelating `Categories(1)` *and* `Products(0)`

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

4.5 Addressing Operations

4.5.1 Addressing Actions

The semantic rules for addressing and invoking actions are defined in the **[OData-Protocol]** document. 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 semantic rules for addressing and invoking functions are defined in the **[OData-Protocol]** document. The grammar for addressing and invoking functions is defined by a number syntax grammar 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 `filter` and `orderby` syntax rules define the grammar for invoking functions 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].

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 Property Value

To address the raw value of a primitive property, clients append a path segment containing the string `$value` to the property URL.

Properties 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.

Example 26: the number of related entities

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

Example 27: the number of entities in an entity set

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

Example 28: 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 29: entity count in an $orderby expression

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

4.9 Addressing Derived Types

Any resource path or path expression identifying a collection of entities or complex type instances can be appended with a path segment containing the qualified name of a type derived from the declared 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 path 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 that is part of a Boolean expression, the type cast will evaluate to `null`.

Example 30: entity set restricted to VipCustomer instances

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

Example 31: 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 32: 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 33: 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 34: 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
```

4.10 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 35: request the media stream for the picture with the key value `Sunset4321299432`:

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

4.11 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 36: 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

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

4.12 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.

Example 37: all entities in a service

```
http://host/service/$all
```

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 prefixed with a dollar (`$`) character.

For `GET` 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 `$expand` and `$select`.
- Resource paths identifying a collection allow `$filter`, `$count`, `$orderby`, `$skip`, and `$top`.
- Resource paths identifying a collection of entities allow `$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 entity sets as well as all `PUT` and `DELETE` requests do not allow system query options.

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 MUST NOT be specified more than once for any resource.

The semantics of all system query options are defined in the **[OData-Protocol]** document.

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 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.

5.1.1.1 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.

Operands of collection, entity, and complex types are not supported in logical operators.

The syntax rules for the logical operators are defined in **[OData-ABNF]**.

The six comparison operators can be used with all primitive values except `Edm.Binary`, `Edm.Stream`, and the `Edm.Geo` types.

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.

The `null` value is equal to itself, and only 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.

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.

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

For Boolean values true is greater than 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.

If only one operand is `null`, the operator returns false. If both operands are `null`, it returns true because `null` is equal to itself.

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.

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.

If only one operand is `null`, the operator returns false. If both operands are `null`, it returns true because `null` is equal to itself.

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 hand 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 Logical Operator Examples

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

Example 38: all products with a `Name` equal to `'Milk'`

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

Example 39: all products with a `Name` not equal to `'Milk'`

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

Example 40: all products with a Name greater than 'Milk':

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

Example 41: all products with a Name greater than or equal to 'Milk':

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

Example 42: all products with a Name less than 'Milk':

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

Example 43: all products with a Name less than or equal to 'Milk':

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

Example 44: all products with the Name 'Milk' that also have a Price less than 2.55:

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

Example 45: all products that either have the Name 'Milk' or have a Price less than 2.55:

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

Example 46: all products that do not have a Name that ends with 'ilk':

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

Example 47: all products whose `style` value includes `Yellow`:

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

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, the result is null.

The syntax rules for the arithmetic operators are defined in **[OData-ABNF]**.

5.1.1.2.1 Addition

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

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 `DateTimeOffset`

5.1.1.2.2 Subtraction

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

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

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.

5.1.1.2.5 Division

The `div` operator divides the left numeric operand by the right numeric operand. If the right operand is zero and the left operand is neither of type `Edm.Single` nor `Edm.Double`, the request fails. If the left operand is of type `Edm.Single` or `Edm.Double`, then positive `div` zero returns `INF`, negative `div` zero returns `–INF`, and zero `div` zero returns `NaN`.

5.1.1.2.6 Modulo

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

5.1.1.2.7 Arithmetic Operator Examples

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

Example 48: all products with a Price of 2.55:

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

Example 49: all products with a Price of 2.55:

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

Example 50: all products with a Price of 2.55:

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

Example 51: all products with a Price of 2.55:

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

Example 52: 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 53: 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 `$filter` or `$orderby` system query options. The following sections describe the available functions. 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`.

The syntax rules for all functions are defined in **[OData-ABNF]**.

5.1.1.4.1 `contains`

The `contains` function has the following signature:

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

The `contains` function returns `true` if the second parameter string value is a substring of the first parameter string value, otherwise it returns `false`. The `containsMethodCallExpr` syntax rule defines how the `contains` function is invoked.

Example 54: all customers with a CompanyName that contains 'Alfreds'

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

5.1.1.4.2 `endswith`

The `endswith` function has the following signature:

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

The `endswith` function returns `true` if the first parameter string value ends with the second parameter string value, otherwise it returns `false`. The `endsWithMethodCallExpr` syntax rule defines how the `endswith` function is invoked.

Example 55: all customers with a CompanyName that ends with 'Futterkiste'

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

5.1.1.4.3 `startswith`

The `startswith` function has the following signature:

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

The `startswith` function returns `true` if the first parameter string value starts with the second parameter string value, otherwise it returns `false`. The `startsWithMethodCallExpr` syntax rule defines how the `startswith` function is invoked.

Example 56: all customers with a CompanyName that starts with 'Alfr'

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

5.1.1.4.4 `length`

The `length` function has the following signature:

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

The `length` function returns the number of characters in the parameter value. The `lengthMethodCallExpr` syntax rule defines how the `length` function is invoked.

Example 57: all customers with a `CompanyName` that is 19 characters long

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

5.1.1.4.5 `indexof`

The `indexof` function has the following signature:

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

The `indexof` function returns the zero-based character position of the first occurrence of the second parameter value in the first parameter value. The `indexOfMethodCallExpr` syntax rule defines how the `indexof` function is invoked.

Example 58: all customers with a `CompanyName` containing '`lfreds`' starting at the second character

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

5.1.1.4.6 `substring`

The `substring` function has consists of two overloads, with the following signatures:

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

The two argument `substring` function 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 argument `substring` function returns a substring of the first parameter string value identified by selecting M characters starting at the Nth character (where N is the second parameter integer value and M is the third parameter integer value).

The `substringMethodCallExpr` syntax rule defines how the `substring` functions are invoked.

Example 59: 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 60: all customers with a `CompanyName` that has '`lf`' as the second and third characters

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

5.1.1.4.7 `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 61: 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'
```

5.1.1.4.8 `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 62: 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'
```

5.1.1.4.9 `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 63: all customers with a `CompanyName` without leading or trailing whitespace characters

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

5.1.1.4.10 `concat`

The `concat` function has the following signature:

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

The `concat` function returns a string that appends the second input parameter string value to the first. The `concatMethodCallExpr` syntax rule defines how the `concat` function is invoked.

Example 64: all customers from Berlin, Germany

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

5.1.1.4.11 `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 (i.e. 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 65: all employees born in 1971

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

5.1.1.4.12 `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 (i.e. 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 66: all employees born in May

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

5.1.1.4.13 `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 (i.e. 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 67: all employees born on the 8th day of a month

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

5.1.1.4.14 `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 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 (i.e. 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 68: all employees born in the 4th hour of a day

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

5.1.1.4.15 `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 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 69: all employees born in the 40th minute of any hour on any day

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

5.1.1.4.16 `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 (without the fractional part) of the `DateTimeOffset` or `TimeOfDay` parameter value. The `secondMethodCallExpr` syntax rule defines how the `second` function is invoked.

Example 70: all employees born in the 40th second of any minute of any hour on any day

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

5.1.1.4.17 `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 71: 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
```

5.1.1.4.18 `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.

5.1.1.4.19 `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 (i.e. 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.

5.1.1.4.20 `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.

5.1.1.4.21 `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, e.g. 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 (i.e., UTC).

5.1.1.4.22 `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.

5.1.1.4.23 `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.

5.1.1.4.24 `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.

5.1.1.4.25 `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 72: all orders with freight costs that round to 32

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

5.1.1.4.26 `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 73: all orders with freight costs that round down to 32

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

5.1.1.4.27 `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 74: all orders with freight costs that round up to 32

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

5.1.1.4.28 `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, 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, otherwise it returns `false`.

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

Example 75: orders that are also `BigOrders`

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

Example 76: orders of a customer that is a `VIPCustomer`

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

5.1.1.4.29 `cast`

The `cast` function has the following signatures:

```
Edm.Any cast(type)
Edm.Any 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 rules:

- The `null` value can be cast to any type.

- 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`.

- Numeric primitive types are cast to each other with appropriate rounding. The cast fails if the integer part doesn't fit into target type.

- `Edm.DateTimeOffset`, `Edm.Duration`, and `Edm.TimeOfDay` values can be cast to the same type with a different precision with appropriate rounding.

- Collections are cast item by item.

- Services MAY support structural casting of entities and complex type instances to a derived type, or arbitrary structured type, by assigning values of identically named properties and casting them recursively. The cast fails if one of the property-value casts fails or the target type contains non-nullable properties that have not been assigned a value.

The `cast` function is optional for primitive values (first three rules).

If the cast fails the `cast` function returns `null`.

5.1.1.4.30 `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.

5.1.1.4.31 `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`.

5.1.1.4.32 `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.

5.1.1.5 Lambda Operators

OData defines two operators that evaluate a Boolean expression on a collection. Both must be prepended with a navigation path that identifies a collection. The argument of a lambda operator is a lambda variable name followed by a colon (`:`) and a Boolean expression that uses the lambda variable name to refer to properties of the related entities identified by the navigation path.

5.1.1.5.1 `any`

The `any` operator applies a Boolean expression to each member of a collection and returns `true` if the expression is `true` for any member of the collection, otherwise it returns `false`. The `any` operator without an argument returns `true` if the collection is not empty.

Example 77: all `Orders` *that have any* `Items` *with a* `Quantity` *greater than* `100`

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

5.1.1.5.2 `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`.

Example 78: all `Orders` *that have only* `Items` *with a* `Quantity` *greater than* `100`

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

5.1.1.6 Literals

5.1.1.6.1 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]**.

Example 79: 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'
TimeOfDayValue eq 07:59:59.999
GuidValue eq 01234567-89ab-cdef-0123-456789abcdef
Int64Value eq 0
ColorEnumValue eq Sales.Pattern'Yellow',
geo.distance(Location,geography'SRID=0;Point(142.1 64.1)')
```

5.1.1.6.2 Complex 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]**.

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

Example 80: collection of string literals

```
http://host/service/ProductsByColor?colors=["red","green"]
```

5.1.1.6.3 `null`

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

5.1.1.6.4 `$it`

The `$it` literal can be used in expressions to refer to the current instance of the collection identified by the resource path. It can be used to compare properties of related entities to properties of the current instance in expressions within lambda operators, for example in `$filter` and `$orderby` expressions on collections of primitive types, or in `$filter` expressions nested within `$expand`.

Example 81: email addresses ending with `.com` assuming `EmailAddresses` is a collection of strings

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

Example 82: 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.

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

5.1.1.6.5 `$root`

The `$root` literal can be used in expressions to refer to resources of the same service. It can be used as a single-valued expression or within complex or collection literals.

Example 83: all employees with the same last name as employee `A1235`

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

Example 84: 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')]
```

5.1.1.7 Path Expressions

Properties and navigation properties of the entity type of the set of resources that are addressed by the request URL 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 85: 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.

5.1.1.8 Parameter Aliases

Parameter aliases can be used within $filter or $orderby in place of expressions that evaluate to a primitive value, a complex value, or a collection of primitive or complex values. Parameter names start with the at sign (@) and can be used in more than one place in the expression. The value for the parameter alias is supplied in a query option with the same name as the parameter.

Example 86:

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

Example 87:

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

5.1.1.9 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
	xxx( )	Method Call	methodCallExpr boolMethodCallExpr functionExpr
Unary	–	Negation	negateExpr
	not	Logical Negation	notExpr
	cast( )	Type Casting	castExpr
Multiplicative	mul	Multiplication	mulExpr
	div	Division	divExpr
	mod	Modulo	modExpr
Additive	add	Addition	addExpr
	sub	Subtraction	subExpr

Group	Operator	Description	ABNF Expression
Relational	gt	Greater Than	gtExpr
	ge	Greater than or Equal	geExpr
	lt	Less Than	ltExpr
	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

5.1.1.10 Numeric Promotion

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.

If the result of an arithmetic operation does not fit into the type determined by the above rules, the next-wider type is used in the above order, with `Edm.Double` considered widest.

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

5.1.2 System Query Option `$expand`

The `$expand` system query option specifies the related resources to be included in line with retrieved resources.

What follows is a (non-normative) snippet from **[OData-ABNF]** that describes the syntax of `$expand`:

```
expand              = '$expand' EQ expandItem *( COMMA expandItem )

expandItem          = STAR [ ref / OPEN levels CLOSE ]
                    / expandPath
                      [ ref   [ OPEN expandRefOption
                             *( SEMI expandRefOption   ) CLOSE ]
                      / count [ OPEN expandCountOption
                             *( SEMI expandCountOption ) CLOSE ]
                      /        OPEN expandOption
                             *( SEMI expandOption      ) CLOSE
                      ]
```

```
expandPath         = [ qualifiedEntityTypeName "/" ]
                     *( ( complexProperty / complexColProperty ) "/"
                        [ qualifiedComplexTypeName "/" ] )
                     navigationProperty
                     [ "/" qualifiedEntityTypeName ]
expandCountOption = filter
                   / search
expandRefOption   = expandCountOption
                   / orderby
                   / skip
                   / top
                   / inlinecount
expandOption      = expandRefOption
                   / select
                   / expand
                   / levels
```

Each `expandItem` is evaluated relative to the entity containing the navigation property being expanded.

A type cast using the `qualifiedEntityTypeName` to a type containing the property is required in order to expand a navigation property defined on a derived type.

An arbitrary number of single- or collection-valued complex properties, optionally followed by a type cast, allow drilling into complex properties.

The `navigationProperty` segment MUST identify a navigation property defined on the entity type of the request, the derived entity type specified in the type cast, or the last complex type identified by the complex property path.

Example 88: expand a navigation property of an entity type

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

Example 89: expand a navigation property of a complex type

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

A navigation property MUST NOT appear in more than one `expandItem`.

Query options can be applied to the expanded navigation property by appending a semicolon-separated list of query options, enclosed in parentheses, to the navigation property name. Allowed system query options are $filter, $select, $orderby, $skip, $top, $count, $search, and $expand.

Example 90: 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)
```

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.

Example 91: all categories and for each category the references of all related products

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

Example 92: 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 93: 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)
```

Cyclic navigation properties (whose target type is identical or can be cast to its source type) can be recursively expanded using the special $levels option. The value of the $levels 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.

Example 94: all employees with their manager, manager's manager, and manager's manager's manager

```
http://host/service/Employees?$expand=Model.Manager/DirectReports($levels=3)
```

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 option. The star operator can be combined with explicitly named navigation properties, which take precedence over the star operator.

Example 95: expand Supplier and include references for all other related entities

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

Example 96: expand all related entities and their related entities

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

5.1.3 System Query Option `$select`

The $select system query option allows clients to requests a limited 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 as a selectItem.

What follows is a (non-normative) snippet from **[OData-ABNF]** showing the syntax of $select:

```
select         = '$select' EQ selectItem *( COMMA selectItem )

selectItem     = STAR
               / allOperationsInSchema
               / [ qualifiedEntityTypeName "/" ]
                 ( selectProperty
                 / qualifiedActionName
                 / qualifiedFunctionName
                 )

selectProperty = primitiveProperty
               / primitiveColProperty
               / navigationProperty
               / selectPath [ "/" selectProperty ]

selectPath     = ( complexProperty / complexColProperty )
                 [ "/" qualifiedComplexTypeName ]
```

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 selectItem in the $select clause indicates that the response MUST include the declared or dynamic properties, actions and functions identified by that selectItem. The simplest form of a selectItem explicitly requests a property defined on the entity type of the resources identified by the resource path section of the URL.

Example 97: 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 98: all structural properties of all products

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

If the `selectItem` is not defined for the type of the resource, and that type is defined as open, then the property is treated as null for all instances on which it is not defined.

If the `selectItem` is not defined for the type of the resource, and that type is not defined as open, then the request is considered malformed.

If the `selectItem` 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 5.1.1.10.

Example 99: name and description of all products, plus name of expanded category

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

The `selectItem` MUST be prefixed with a `qualifiedEntityTypeName` or `qualifiedComplexTypeName` in order to select a property defined on a type derived from the type of the resource segment.

A `selectItem` 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 100: 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
```

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

If any `selectItem` (including a star) is specified, actions and functions SHOULD be omitted unless explicitly requested using a `qualifiedActionName`, a `qualifiedFunctionName` or the `allOperationsInSchema`.

If an action or function is requested as a `selectItem`, either explicitly by using a `qualifiedActionName` or `qualifiedFunctionName` cause, or implicitly by using `allOperationsInSchema`, 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 `selectItem` using a `qualifiedActionName` or a `qualifiedFunctionName` and that operation cannot be bound to the entities requested, the service MUST ignore the `selectItem`.

Example 101: 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 `selectItems` 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 `selectItem`.

If a `selectItem` is a path expression requesting a component of a complex property and the complex property is `null` on an instance, then the component is treated as `null` as well.

5.1.4 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 the **[OData-Protocol]** document.

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

5.1.5 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 the **[OData-Protocol]** document. The **[OData-ABNF]** `top` and `skip` syntax rules define the formal grammar of the `$top` and `$skip` query options respectively.

5.1.6 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 the **[OData-Protocol]** document.

5.1.7 System Query Option `$search`

The `$search` system query option allows clients to request entities matching a free-text search expression.

The `$search` query option can be applied to a URL representing a collection of entities to return all matching entities within the collection. Applying the `$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 entities that match both criteria.

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

Example 102: 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
```

5.1.7.1 Search Expressions

Search expressions are used within the `$search` system query option to request entities matching the specified 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 of the expressions evaluate to true, otherwise false.

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

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

5.1.8 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 the **[OData-Protocol]** document.

The **[OData-ABNF]** format syntax rule define the formal grammar of the $format 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 103: 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 function parameters or within a $filter or $orderby expression.

Parameter aliases MUST start with an @ character.

The semantics of parameter aliases are covered in **[OData-Protocol]**.

The **[OData-ABNF]** rule aliasAndValue defines the formal grammar for passing parameter aliases as query options.

6 Conformance

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

Appendix A. Acknowledgments

The contributions of the OASIS OData Technical Committee members, enumerated in **[OData-Protocol]**, are gratefully acknowledged.

Appendix B. Revision History

Revision	Date	Editor	Changes Made
Working Draft 01	2012-08-22	Michael Pizzo	Translated Contribution to OASIS format/template
Committee Specification Draft 01	2013-04-26	Ralf Handl Michael Pizzo Martin Zurmuehl	Added Full-Text Search, modified expand syntax, expand options, crosstabs, enumerations Fleshed out descriptions and examples and addressed numerous editorial and technical issues processed through the TC Added Conformance section
Committee Specification Draft 02	2013-07-01	Ralf Handl Michael Pizzo Martin Zurmuehl	Described which query options are applicable to which resource types and HTTP methods Simplified URL syntax Extended expand with a STAR operator Added special resources for cross-service search, cross joins, resolution of entity-ids Described handling of null values, division by zero, and overflow in arithmetic operations Added filtering for collections of complex and primitive types
Committee Specification 01	2013-07-30	Michael Pizzo, Ralf Handl, Martin Zurmuehl	Non-Material Changes
Committee Specification Draft 03	2013-10-03	Michael Pizzo, Ralf Handl, Martin Zurmuehl	Accessing properties of derived types Examples for primitive literals Precedence of `has` operator