OData Common Schema Definition Language (CSDL) JSON Representation Version 4.01

Candidate OASIS Standard 01

28 January 2020

This stage:
https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/cos01/odata-csdl-json-v4.01-cos01.docx (Authoritative)
https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/cos01/odata-csdl-json-v4.01-cos01.html
https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/cos01/odata-csdl-json-v4.01-cos01.pdf

Previous stage:
https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/cs02/odata-csdl-json-v4.01-cs02.docx (Authoritative)
https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/cs02/odata-csdl-json-v4.01-cs02.html
https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/cs02/odata-csdl-json-v4.01-cs02.pdf

Latest stage:
https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/odata-csdl-json-v4.01.docx (Authoritative)
https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/odata-csdl-json-v4.01.html
https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/odata-csdl-json-v4.01.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
Martin Zurmuehl (martin.zurmuehl@sap.com), SAP SE

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

- JSON schemas; OData CSDL JSON schema. https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/cos01/schemas/.

Related work:
This specification is related to:

- *OData Version 4.01*. Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. A multi-part Work Product which includes:
  - *OData Version 4.01. Part 1: Protocol*. Latest stage: https://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part1-protocol.html.
  - *OData Version 4.01. Part 2: URL Conventions*. Latest stage: https://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part2-url-conventions.html.

    - *ABNF components: OData ABNF Construction Rules Version 4.01 and OData ABNF Test Cases*. https://docs.oasis-open.org/odata/odata/v4.01/cos01/abnf/.
- *OData Common Schema Definition Language (CSDL) XML Representation Version 4.01.* Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. Latest stage: https://docs.oasis-open.org/odata/odata-csdl-xml/v4.01/odata-csdl-xml-v4.01.html.
- *OData Vocabularies Version 4.0.* Edited by Michael Pizzo, Ralf Handl, and Ram Jeyaraman. Latest stage: http://docs.oasis-open.org/odata/odata-vocabularies/v4.0/odata-vocabularies-v4.0.html.
- *OData JSON Format Version 4.01*. Edited by Michael Pizzo, Ralf Handl, and Mark Biamonte. Latest stage: https://docs.oasis-open.org/odata/odata-json-format/v4.01/odata-json-format-v4.01.html.

Abstract:
OData services are described by an Entity Model (EDM). The Common Schema Definition Language (CSDL) defines specific representations of the entity data model exposed by an OData service, using XML, JSON, and other formats. This document (OData CSDL JSON Representation) specifically defines the JSON representation of CSDL.

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://www.oasis-open.org/committees/tc_home.php?wg_abbrev=odata#technical.

TC members should send comments on this specification to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "Send A Comment" button on the TC's web page at https://www.oasis-open.org/committees/odata/.

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.

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

[OData-CSDL-JSON-v4.01]

OData Common Schema Definition Language (CSDL) JSON Representation Version 4.01. Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. 28 January 2020. Candidate OASIS Standard 01. https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/cos01/odata-csdl-json-v4.01-cos01.html. Latest stage: https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/odata-csdl-json-v4.01.html.

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.

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 https://www.oasis-open.org/policies-guidelines/trademark for above guidance.

Table of Contents

1 Introduction

OData services are described in terms of an Entity Model. The Common Schema Definition Language (CSDL) defines a representation of the entity model exposed by an OData service using the JavaScript Object Notation (JSON), see **[RFC8259]**.

This format is based on the OpenUI5 OData V4 Metadata JSON Format, see **[OpenUI5]**, with some extensions and modifications made necessary to fully cover OData CSDL Version 4.01.

1.1 IPR Policy

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).

1.2 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]**.

1.3 Normative References

[ECMAScript] *ECMAScript 2016 Language Specification, 7th Edition,* June 2016. Standard ECMA-262. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[EPSG] European Petroleum Survey Group (EPSG). http://www.epsg.org/.

[OData-ABNF] *OData ABNF Construction Rules Version 4.01*.
See link in "Additional artifacts" section on cover page.

[OData-CSDL-Schema] *OData CSDL JSON Schema.*
See link in "Additional artifacts" section on cover page.

[OData-CSDLXML] OData Common Schema Definition Language (CSDL) XML Representation Version 4.01.
See link in "Related work" section on cover page.

[OData-JSON] *OData JSON Format Version 4.01.*
See link in "Related work" section on cover page.

[OData-Protocol] *OData Version 4.01 Part 1: Protocol*.
See link in "Related work" section on cover page.

[OData-URL] *OData Version 4.01 Part 2: URL Conventions*.
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.

[OData-VocMeasures] *OData Vocabularies Version 4.0: Measures Vocabulary.*
See link in "Related work" section on cover page.

[OData-VocValidation] *OData Vocabularies Version 4.0: Validation 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, March 1997. https://tools.ietf.org/html/rfc2119.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, March 2012. http://tools.ietf.org/html/rfc6570.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC7493, March 2015. https://tools.ietf.org/html/rfc7493.

[RFC8259] Bray, T., Ed., “The JavaScript Object Notation (JSON) Data Interchange Format”, RFC 8259, December 2017. http://tools.ietf.org/html/rfc8259.

[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/.

1.4 Non-Normative References

[OpenUI5] OpenUI5 Version 1.40.10 – OData V4 Metadata JSON Format, https://openui5.hana.ondemand.com/1.40.10/#docs/guide/87aac894a40640f89920d7b2a414499b.html

1.5 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.

Representation-specific text is indented and marked with vertical lines.

> ### Representation-Specific Headline
>
> Normative representation-specific text

All other text is normative unless otherwise labeled.

2 JSON Representation

OData CSDL JSON is a full representation of the OData Common Schema Definition Language in the JavaScript Object Notation (JSON) defined in **[RFC8259]**. It additionally follows the rules for "Internet JSON" (I-JSON) defined in **[RFC7493]** for e.g. objects, numbers, date values, and duration values.

It is an alternative to the CSDL XML representation defined in **[OData-CSDLXML]** and neither adds nor removes features.

2.1 Requesting the JSON Representation

The OData CSDL JSON representation can be requested using the `$format` query option in the request URL with the media type `application/json`, optionally followed by media type parameters, or the case-insensitive abbreviation `json` which MUST NOT be followed by media type parameters.

Alternatively, this representation can be requested using the `Accept` header with the media type `application/json`, optionally followed by media type parameters.

If specified, `$format` overrides any value specified in the `Accept` header.

The response MUST contain the `Content-Type` header with a value of `application/json`, optionally followed by media type parameters.

Possible media type parameters are:

- `IEEE754Compatible`
- `metadata`

The names and values of these parameters are case-insensitive.

2.1.1 Controlling the Representation of Numbers

The `IEEE754Compatible=true` parameter indicates that the service MUST serialize `Edm.Int64` and `Edm.Decimal` numbers as strings. This is in conformance with **[RFC7493]**. If not specified, or specified as `IEEE754Compatible=false`, all numbers MUST be serialized as JSON numbers.

This enables support for JavaScript numbers that are defined to be 64-bit binary format IEEE 754 values **[ECMAScript]** (see section 4.3.1.9) resulting in integers losing precision past 15 digits, and decimals losing precision due to the conversion from base 10 to base 2.

Responses that format `Edm.Int64` and `Edm.Decimal` values as strings MUST specify this parameter in the media type returned in the `Content-Type` header.

2.1.2 Controlling the Amount of Control Information

The representation of constant annotation values in CSDL JSON documents closely follows the representation of data defined in **[OData-JSON]**.

A client application can use the `metadata` format parameter in the `Accept` header when requesting a CSDL JSON document to influence how much control information will be included in the response.

Other `Accept` header parameters are orthogonal to the `metadata` parameter and are therefore not mentioned in this section.

2.1.2.1 `metadata=minimal`

The `metadata=minimal` format parameter indicates that the service SHOULD remove computable control information from the payload wherever possible.

This means that the `@type` control information is only included if the type of the containing object or targeted property cannot be heuristically determined, e.g. for

- Terms or term properties with an abstract declared type,

- Terms or term properties with a declared type that has derived types, or
- Dynamic properties of open types.

See **[OData-JSON]** for the exact rules.

2.1.2.2 metadata=full

The `metadata=full` format parameter indicates that the service MUST include all control information explicitly in the payload.

This means that the `@type` control information is included in annotation values except for primitive values whose type can be heuristically determined from the representation of the value, see **[OData-JSON]** for the exact rules.

2.1.2.3 metadata=none

The `metadata=none` format parameter indicates that the service SHOULD omit all control information.

2.2 Design Considerations

CSDL JSON documents are designed for easy and efficient lookup of model constructs by their name without having to know or guess what kind of model element it is. Thus, all primary model elements (entity types, complex types, type definitions, enumeration types, terms, actions, functions, and the entity container) are direct members of their schema, using the schema-unique name as the member name. Similarly, child elements of primary model elements (properties, navigation properties, enumeration type members, entity sets, singletons, action imports, and function imports) are direct members of the objects describing their parent model element, using their locally unique name as the member name.

To avoid name collisions, all fixed member names are prefixed with a dollar (`$`) sign and otherwise have the same name and capitalization as their counterparts in the CSDL XML representation **[OData-CSDLXML]** (with one exception: the counterpart of the `EntitySet` element's `EntityType` attribute is `$Type`, to harmonize it with all other type references).

Additional fixed members introduced by this specification and without counterpart in **[OData-CSDLXML]** are also prefixed with a dollar (`$`) sign and use upper-camel-case names. One of these is `$Kind` which represents the kind of model element. Its value is the upper-camel-case local name of the XML element representing this kind of model element in **[OData-CSDLXML]**, e.g. `EntityType` or `NavigationProperty`.

While the XML representation of CSDL allows referencing model elements with alias-qualified names as well as with namespace-qualified names, this JSON representation requires the use of alias-qualified names if an alias is specified for an included or document-defined schema. Aliases are usually shorter than namespaces, so this reduces text size of the JSON document. Text size matters even if the actual HTTP messages are sent in compressed form because the decompressed form needs to be reconstructed, and clients not using a streaming JSON parser have to materialize the full JSON document before parsing.

To further reduce size the member `$Kind` is optional for structural properties as these are more common than navigation properties, and the member `$Type` is optional for string properties, parameters, and return types, as this type is more common than other primitive types.

In general, all members that have a default value SHOULD be omitted if they have the default value.

2.3 JSON Schema Definition

The structure of CSDL JSON documents can be verified with the JSON Schema **[OData-CSDL-Schema]** provided as an additional artifact of this prose specification. This schema only defines the shape of a well-formed CSDL JSON document but is not descriptive enough to define what a correct CSDL JSON document MUST be in every imaginable use case. This specification document defines additional rules that correct CSDL JSON documents MUST fulfill. In case of doubt on what makes a CSDL JSON document correct the rules defined in this specification document take precedence.

3 Entity Model

An OData service exposes a single entity model. This model may be distributed over several schemas, and these schemas may be distributed over several documents.

A service is defined by a single CSDL document which can be accessed by sending a `GET` request to `<serviceRoot>/$metadata`. This document is called the metadata document. It MAY reference other CSDL documents.

The metadata document contains a single entity container that defines the resources exposed by this service. This entity container MAY extend an entity container defined in a referenced document.

The *model* of the service consists of all CSDL constructs used in its entity containers.

The *scope* of a CSDL document is the document itself and all schemas included from directly referenced documents. All entity types, complex types and other named model elements *in scope* (that is, defined in the document itself or a schema of a directly referenced document) can be accessed from a referencing document by their qualified names. This includes the built-in primitive and abstract types.

Referencing another document may alter the model defined by the referencing document. For instance, if a referenced document defines an entity type derived from an entity type in the referencing document, then an entity set of the service defined by the referencing document may return entities of the derived type. This is identical to the behavior if the derived type had been defined directly in the referencing document.

Note: referencing documents is not recursive. Only named model elements defined in directly referenced documents can be used within the schema. However, those elements may in turn include or reference model elements defined in schemas referenced by their defining schema.

3.1 Nominal Types

A nominal type has a name that MUST be a simple identifier. Nominal types are referenced using their qualified name. The qualified type name MUST be unique within a model as it facilitates references to the element from other parts of the model.

Names are case-sensitive, but service authors SHOULD NOT choose names that differ only in case.

3.2 Structured Types

Structured types are composed of other model elements. Structured types are common in entity models as the means of representing entities and structured properties in an OData service. Entity types and complex types are both structured types.

Structured Types are composed of zero or more structural properties and navigation properties.

Open entity types and open complex types allow properties to be added dynamically to instances of the open type.

3.3 Primitive Types

Structured types are composed of other structured types and primitive types. OData defines the following primitive types:

Type	Meaning
`Edm.Binary`	Binary data
`Edm.Boolean`	Binary-valued logic
`Edm.Byte`	Unsigned 8-bit integer
`Edm.Date`	Date without a time-zone offset

Type	Meaning
`Edm.DateTimeOffset`	Date and time with a time-zone offset, no leap seconds
`Edm.Decimal`	Numeric values with decimal representation
`Edm.Double`	IEEE 754 binary64 floating-point number (15-17 decimal digits)
`Edm.Duration`	Signed duration in days, hours, minutes, and (sub)seconds
`Edm.Guid`	16-byte (128-bit) unique identifier
`Edm.Int16`	Signed 16-bit integer
`Edm.Int32`	Signed 32-bit integer
`Edm.Int64`	Signed 64-bit integer
`Edm.SByte`	Signed 8-bit integer
`Edm.Single`	IEEE 754 binary32 floating-point number (6-9 decimal digits)
`Edm.Stream`	Binary data stream
`Edm.String`	Sequence of characters
`Edm.TimeOfDay`	Clock time 00:00-23:59:59.999999999999
`Edm.Geography`	Abstract base type for all Geography types
`Edm.GeographyPoint`	A point in a round-earth coordinate system
`Edm.GeographyLineString`	Line string in a round-earth coordinate system
`Edm.GeographyPolygon`	Polygon in a round-earth coordinate system
`Edm.GeographyMultiPoint`	Collection of points in a round-earth coordinate system
`Edm.GeographyMultiLineString`	Collection of line strings in a round-earth coordinate system
`Edm.GeographyMultiPolygon`	Collection of polygons in a round-earth coordinate system
`Edm.GeographyCollection`	Collection of arbitrary Geography values
`Edm.Geometry`	Abstract base type for all Geometry types
`Edm.GeometryPoint`	Point in a flat-earth coordinate system
`Edm.GeometryLineString`	Line string in a flat-earth coordinate system
`Edm.GeometryPolygon`	Polygon in a flat-earth coordinate system
`Edm.GeometryMultiPoint`	Collection of points in a flat-earth coordinate system
`Edm.GeometryMultiLineString`	Collection of line strings in a flat-earth coordinate system
`Edm.GeometryMultiPolygon`	Collection of polygons in a flat-earth coordinate system
`Edm.GeometryCollection`	Collection of arbitrary Geometry values

`Edm.Date` and `Edm.DateTimeOffset` follow **[XML-Schema-2]** and use the proleptic Gregorian calendar, allowing the year `0000` (equivalent to 1 BCE) and negative years (year `-0001` being equivalent to 2 BCE etc.). The supported date range is service-specific and typically depends on the underlying persistency layer, e.g. SQL only supports years `0001` to `9999`.

`Edm.Decimal` with a `Scale` value of `floating`, `Edm.Double`, and `Edm.Single` allow the special numeric values `-INF`, `INF`, and `NaN`.

`Edm.Stream` is a primitive type that can be used as a property of an entity type or complex type, the underlying type for a type definition, or the binding parameter or return type of an action or function. `Edm.Stream`, or a type definition whose underlying type is `Edm.Stream`, cannot be used in collections or for non-binding parameters to functions or actions.

Some of these types allow facets, defined in section "Type Facets".

See rule `primitiveLiteral` in **[OData-ABNF]** for the representation of primitive type values in URLs and **[OData-JSON]** for the representation in requests and responses.

3.4 Built-In Abstract Types

The following built-in abstract types can be used within a model:

- `Edm.PrimitiveType`
- `Edm.ComplexType`
- `Edm.EntityType`
- `Edm.Untyped`

Conceptually, these are the abstract base types for primitive types (including type definitions and enumeration types), complex types, entity types, or any type or collection of types, respectively, and can be used anywhere a corresponding concrete type can be used, except:

- `Edm.EntityType`
  - cannot be used as the type of a singleton in an entity container because it doesn't define a structure, which defeats the purpose of a singleton.
  - cannot be used as the type of an entity set because all entities in an entity set must have the same key fields to uniquely identify them within the set.
  - cannot be the base type of an entity type or complex type.
- `Edm.ComplexType`
  - cannot be the base type of an entity type or complex type.
- `Edm.PrimitiveType`
  - cannot be used as the type of a key property of an entity type or as the underlying type of an enumeration type.
  - cannot be used as the underlying type of a type definition in a CSDL document with a version of `4.0`.
  - can be used as the underlying type of a type definition in a CSDL document with a version of `4.01` or greater.
- `Edm.Untyped`
  - cannot be returned in a payload with an `OData-Version` header of `4.0`. Services should treat untyped properties as dynamic properties in `4.0` payloads.
  - cannot be used as the type of a key property of an entity type.
  - cannot be the base type of an entity type or complex type.
  - cannot be used as the underlying type of a type definition or enumeration type.
- `Collection(Edm.PrimitiveType)`
  - cannot be used as the type of a property or term.
  - cannot be used as the type of a parameter or the return type of an action or function.
- `Collection(Edm.Untyped)`

- cannot be returned in a payload with an `OData-Version` header of `4.0`. Services should treat untyped properties as dynamic properties in `4.0` payloads.

3.5 Built-In Types for defining Vocabulary Terms

Vocabulary terms can, in addition, use

- `Edm.AnnotationPath`
- `Edm.PropertyPath`
- `Edm.NavigationPropertyPath`
- `Edm.AnyPropertyPath` (`Edm.PropertyPath` or `Edm.NavigationPropertyPath`)
- `Edm.ModelElementPath` (any model element, including `Edm.AnnotationPath`, `Edm.NavigationPropertyPath`, and `Edm.PropertyPath`)

as the type of a primitive term, or the type of a property of a complex type (recursively) that is exclusively used as the type of a term. See section "Path Expressions" for details.

3.6 Annotations

Many parts of the model can be decorated with additional information using annotations. Annotations are identified by their term name and an optional qualifier that allows applying the same term multiple times to the same model element.

A model element MUST NOT specify more than one annotation for a given combination of term and qualifier.

4 CSDL JSON Document

Document Object

A CSDL JSON document consists of a single JSON object. This document object MUST contain the member `$Version`.

The document object MAY contain the member `$Reference` to reference other CSDL documents.

It also MAY contain members for schemas.

If the CSDL JSON document is the metadata document of an OData service, the document object MUST contain the member `$EntityContainer`.

$Version

The value of `$Version` is a string containing either `4.0` or `4.01`.

$EntityContainer

The value of `$EntityContainer` is value is the namespace-qualified name of the entity container of that service. This is the only place where a model element MUST be referenced with its namespace-qualified name and use of the alias-qualified name is not allowed.

Example 2:

```
{
  "$Version": "4.01",
  "$EntityContainer": "org.example.DemoService",
  …
}
```

4.1 Reference

A reference to an external CSDL document allows to bring part of the referenced document's content into the scope of the referencing document.

A reference MUST specify a URI that uniquely identifies the referenced document, so two references MUST NOT specify the same URI. The URI SHOULD be a URL that locates the referenced document. If the URI is not dereferencable it SHOULD identify a well-known schema. The URI MAY be absolute or relative URI; relative URLs are relative to the URL of the document containing the reference, or relative to a base URL specified in a format-specific way.

A reference MAY be annotated.

The `Core.SchemaVersion` annotation, defined in **[OData-VocCore]**, MAY be used to indicate a particular version of the referenced document. If the `Core.SchemaVersion` annotation is present, the `$schemaversion` system query option, defined **[OData-Protocol]**, SHOULD be used when retrieving the referenced schema document.

$Reference

The value of `$Reference` is an object that contains one member per referenced CSDL document. The name of the pair is a URI for the referenced document. The URI MAY be relative to the document containing the `$Reference`. The value of each member is a reference object.

Reference Object

The reference object MAY contain the members `$Include` and `$IncludeAnnotations` as well as annotations.

Example 3: references to other CSDL documents

```
{
  …
  "$Reference": {
    "http://vocabs.odata.org/capabilities/v1": {
      …
    },
    "http://vocabs.odata.org/core/v1": {
      …
    },
    "http://example.org/display/v1": {
      …
    }
  },
  …
}
```

4.2 Included Schema

A reference MAY include zero or more schemas from the referenced document.

The included schemas are identified via their namespace. The same namespace MUST NOT be included more than once, even if it is declared in more than one referenced document.

When including a schema, a simple identifier value MAY be specified as an alias for the schema that is used in qualified names instead of the namespace. For example, an alias of `display` might be assigned to the namespace `org.example.vocabularies.display`. An alias-qualified name is resolved to a fully qualified name by examining aliases for included schemas and schemas defined within the document.

> If an included schema specifies an alias, the alias MUST be used in qualified names throughout the document to identify model elements of the included schema. A mixed use of namespace-qualified names and alias-qualified names is not allowed.

Aliases are document-global, so all schemas defined within or included into a document MUST have different aliases, and aliases MUST differ from the namespaces of all schemas defined within or included into a document.

The alias MUST NOT be one of the reserved values `Edm`, `odata`, `System`, or `Transient`.

An alias is only valid within the document in which it is declared; a referencing document may define its own aliases for included schemas.

> ### `$Include`
>
> The value of `$Include` is an array. Array items are objects that MUST contain the member `$Namespace` and MAY contain the member `$Alias`.
>
> The item objects MAY contain annotations.
>
> ### `$Namespace`
>
> The value of `$Namespace` is a string containing the namespace of the included schema.
>
> ### `$Alias`
>
> The value of `$Alias` is a string containing the alias for the included schema.

Example 4: references to entity models containing definitions of vocabulary terms

```
{
  …
  "$Reference": {
    "http://vocabs.odata.org/capabilities/v1": {
      "$Include": [
        {
```

```
          "$Namespace": "Org.OData.Capabilities.V1",
          "$Alias": "Capabilities"
        }
      ]
    },
    "http://vocabs.odata.org/core/v1": {
      "$Include": [
        {
          "$Namespace": "Org.OData.Core.V1",
          "$Alias": "Core",
          "@Core.DefaultNamespace": true
        }
      ]
    },
    "http://example.org/display/v1": {
      "$Include": [
        {
          "$Namespace": "org.example.display",
          "$Alias": "UI"
        }
      ]
    }
  },
  …
}
```

4.3 Included Annotations

In addition to including whole schemas with all model constructs defined within that schema, annotations can be included with more flexibility.

Annotations are selectively included by specifying the namespace of the annotations' term. Consumers can opt not to inspect the referenced document if none of the term namespaces is of interest for the consumer.

In addition, the qualifier of annotations to be included MAY be specified. For instance, a service author might want to supply a different set of annotations for various device form factors. If a qualifier is specified, only those annotations from the specified term namespace with the specified qualifier (applied to a model element of the target namespace, if present) SHOULD be included. If no qualifier is specified, all annotations within the referenced document from the specified term namespace (taking into account the target namespace, if present) SHOULD be included.

The qualifier also provides consumers insight about what qualifiers are present in the referenced document. If the consumer is not interested in that particular qualifier, the consumer can opt not to inspect the referenced document.

In addition, the namespace of the annotations' target MAY be specified. If a target namespace is specified, only those annotations which apply a term form the specified term namespace to a model element of the target namespace (with the specified qualifier, if present) SHOULD be included. If no target namespace is specified, all annotations within the referenced document from the specified term namespace (taking into account the qualifier, if present) SHOULD be included.

The target namespace also provides consumers insight about what namespaces are present in the referenced document. If the consumer is not interested in that particular target namespace, the consumer can opt not to inspect the referenced document.

> **`$IncludeAnnotations`**
>
> The value of `$IncludeAnnotations` is an array. Array items are objects that MUST contain the member `$TermNamespace` and MAY contain the members `$Qualifier` and `$TargetNamespace`.

$TermNamespace

The value of `$TermNamespace` is a namespace.

$Qualifier

The value of `$Qualifier` is a simple identifier.

$TargetNamespace

The value of `$TargetNamespace` is a namespace.

Example 5: reference documents that contain annotations

```
{
  …
  "$Reference": {
    "http://odata.org/ann/b": {
      "$IncludeAnnotations": [
        {
          "$TermNamespace": "org.example.validation"
        },
        {
          "$TermNamespace": "org.example.display",
          "$Qualifier": "Tablet"
        },
        {
          "$TermNamespace": "org.example.hcm",
          "$TargetNamespace": "com.example.Sales"
        },
        {
          "$TermNamespace": "org.example.hcm",
          "$Qualifier": "Tablet",
          "$TargetNamespace": "com.example.Person"
        }
      ]
    }
  },
  …
}
```

The following annotations from `http://odata.org/ann/b` are included:

- *Annotations that use a term from the `org.example.validation` namespace, and*
- *Annotations that use a term from the `org.example.display` namespace and specify a `Tablet` qualifier and*
- *Annotations that apply a term from the `org.example.hcm` namespace to an element of the `com.example.Sales` namespace and*
- *Annotations that apply a term from the `org.example.hcm` namespace to an element of the `com.example.Person` namespace and specify a `Tablet` qualifier.*

5 Schema

One or more schemas describe the entity model exposed by an OData service. The schema acts as a namespace for elements of the entity model such as entity types, complex types, enumerations and terms.

A schema is identified by a namespace. Schema namespaces MUST be unique within the scope of a document and SHOULD be globally unique. A schema cannot span more than one document.

The schema's namespace is combined with the name of elements in the schema to create unique qualified names, so identifiers that are used to name types MUST be unique within a namespace to prevent ambiguity.

Names are case-sensitive, but service authors SHOULD NOT choose names that differ only in case.

The namespace MUST NOT be one of the reserved values `Edm`, `odata`, `System`, or `Transient`.

> ### Schema Object
>
> A schema is represented as a member of the document object whose name is the schema namespace. Its value is an object that MAY contain the members `$Alias` and `$Annotations`.
>
> The schema object MAY contain members representing entity types, complex types, enumeration types, type definitions, actions, functions, terms, and an entity container.
>
> The schema object MAY also contain annotations that apply to the schema itself.

5.1 Alias

A schema MAY specify an alias which MUST be a simple identifier.

> If a schema specifies an alias, the alias MUST be used instead of the namespace within qualified names throughout the document to identify model elements of that schema. A mixed use of namespace-qualified names and alias-qualified names is not allowed.

Aliases are document-global, so all schemas defined within or included into a document MUST have different aliases, and aliases MUST differ from the namespaces of all schemas defined within or included into a document. Aliases defined by a schema can be used throughout the containing document and are not restricted to the schema that defines them.

The alias MUST NOT be one of the reserved values `Edm`, `odata`, `System`, or `Transient`.

> **`$Alias`**
>
> The value of `$Alias` is a string containing the alias for the schema.

Example 6: document defining a schema `org.example` with an alias and a description for the schema

```
{
  …
  "org.example": {
    "$Alias": "self",
    "@Core.Description": "Example schema",
    …
  },
  …
}
```

5.2 Annotations with External Targeting

$Annotations

The value of `$Annotations` is an object with one member per annotation target. The member name is a path identifying the annotation target, the member value is an object containing annotations for that target.

Example 7: annotations targeting the `Person` type with qualifier `Tablet`

```
"org.example": {
  "$Alias": "self",
  "$Annotations": {
    "self.Person": {
      "@Core.Description#Tablet": "Dummy",
      …
    }
  }
},
```

6 Entity Type

Entity types are nominal structured types with a key that consists of one or more references to structural properties. An entity type is the template for an entity: any uniquely identifiable record such as a customer or order.

The entity type's name is a simple identifier that MUST be unique within its schema.

An entity type can define two types of properties. A structural property is a named reference to a primitive, complex, or enumeration type, or a collection of primitive, complex, or enumeration types. A navigation property is a named reference to another entity type or collection of entity types.

All properties MUST have a unique name within an entity type. Properties MUST NOT have the same name as the declaring entity type. They MAY have the same name as one of the direct or indirect base types or derived types.

Entity Type Object

An entity type is represented as a member of the schema object whose name is the unqualified name of the entity type and whose value is an object.

The entity type object MUST contain the member `$Kind` with a string value of `EntityType`.

It MAY contain the members `$BaseType`, `$Abstract`, `$OpenType`, `$HasStream`, and `$Key`.

It also MAY contain members representing structural properties and navigation properties as well as annotations.

Example 8: a simple entity type

```
"Employee": {
  "$Kind": "EntityType",
  "$Key": [
    "ID"
  ],
  "ID": {},
  "FirstName": {},
  "LastName": {},
  "Manager": {
    "$Kind": "NavigationProperty",
    "$Nullable": true,
    "$Type": "self.Manager"
  }
}
```

6.1 Derived Entity Type

An entity type can inherit from another entity type by specifying it as its base type.

An entity type inherits the key as well as structural and navigation properties of its base type.

An entity type MUST NOT introduce an inheritance cycle by specifying a base type.

$BaseType

The value of `$BaseType` is the qualified name of the base type.

Example 9: a derived entity type based on the previous example

```
"Manager": {
  "$Kind": "EntityType",
  "$BaseType": "self.Employee",
  "AnnualBudget": {
    "$Nullable": true,
    "$Type": "Edm.Decimal",
    "$Scale": 0
```

```
  },
  "Employees": {
    "$Kind": "NavigationProperty",
    "$Collection": true,
    "$Type": "self.Employee"
  }
}
```

Note: the derived type has the same name as one of the properties of its base type.

6.2 Abstract Entity Type

An entity type MAY indicate that it is abstract and cannot have instances.

For OData 4.0 responses a non-abstract entity type MUST define a key or derive from a base type with a defined key.

An abstract entity type MUST NOT inherit from a non-abstract entity type.

> **`$Abstract`**
>
> The value of `$Abstract` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

6.3 Open Entity Type

An entity type MAY indicate that it is open and allows clients to add properties dynamically to instances of the type by specifying uniquely named property values in the payload used to insert or update an instance of the type.

An entity type derived from an open entity type MUST indicate that it is also open.

Note: structural and navigation properties MAY be returned by the service on instances of any structured type, whether or not the type is marked as open. Clients MUST always be prepared to deal with additional properties on instances of any structured type, see **[OData-Protocol]**.

> **`$OpenType`**
>
> The value of `$OpenType` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

6.4 Media Entity Type

An entity type that does not specify a base type MAY indicate that it is a media entity type. *Media entities* are entities that represent a media stream, such as a photo. Use a media entity if the out-of-band stream is the main topic of interest and the media entity is just additional structured information attached to the stream. Use a normal entity with one or more properties of type `Edm.Stream` if the structured data of the entity is the main topic of interest and the stream data is just additional information attached to the structured data. For more information on media entities see **[OData-Protocol]**.

An entity type derived from a media entity type MUST indicate that it is also a media entity type.

Media entity types MAY specify a list of acceptable media types using an annotation with term `Core.AcceptableMediaTypes`, see **[OData-VocCore]**.

> **`$HasStream`**
>
> The value of `$HasStream` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

6.5 Key

An entity is uniquely identified within an entity set by its key. A key MAY be specified if the entity type does not specify a base type that already has a key declared.

In order to be specified as the type of an entity set or a collection-valued containment navigation property, the entity type MUST either specify a key or inherit its key from its base type.

In OData 4.01 responses entity types used for singletons or single-valued navigation properties do not require a key. In OData 4.0 responses entity types used for singletons or single-valued navigation properties MUST have a key defined.

An entity type (whether or not it is marked as abstract) MAY define a key only if it doesn't inherit one.

An entity type's key refers to the set of properties whose values uniquely identify an instance of the entity type within an entity set. The key MUST consist of at least one property.

Key properties MUST NOT be nullable and MUST be typed with an enumeration type, one of the following primitive types, or a type definition based on one of these primitive types:

- `Edm.Boolean`
- `Edm.Byte`
- `Edm.Date`
- `Edm.DateTimeOffset`
- `Edm.Decimal`
- `Edm.Duration`
- `Edm.Guid`
- `Edm.Int16`
- `Edm.Int32`
- `Edm.Int64`
- `Edm.SByte`
- `Edm.String`
- `Edm.TimeOfDay`

Key property values MAY be language-dependent, but their values MUST be unique across all languages and the entity ids (defined in **[OData-Protocol]**) MUST be language independent.

A key property MUST be a non-nullable primitive property of the entity type itself, including non-nullable primitive properties of non-nullable single-valued complex properties, recursively.

In OData 4.01 the key properties of a directly related entity type MAY also be part of the key if the navigation property is single-valued and not nullable. This includes navigation properties of non-nullable single-valued complex properties (recursively) of the entity type. If a key property of a related entity type is part of the key, all key properties of the related entity type MUST also be part of the key.

If the key property is a property of a complex property (recursively) or of a directly related entity type, the key MUST specify an alias for that property that MUST be a simple identifier and MUST be unique within the set of aliases, structural and navigation properties of the declaring entity type and any of its base types.

An alias MUST NOT be defined if the key property is a primitive property of the entity type itself.

For key properties that are a property of a complex or navigation property, the alias MUST be used in the key predicate of URLs instead of the path to the property because the required percent-encoding of the forward slash separating segments of the path to the property would make URL construction and parsing rather complicated. The alias MUST NOT be used in the query part of URLs, where paths to properties don't require special encoding and are a standard constituent of expressions anyway.

$Key

The value of `$Key` is an array with one item per key property.

Key properties without a key alias are represented as strings containing the property name.

Key properties with a key alias are represented as objects with one member whose name is the key alias and whose value is a string containing the path to the property.

Example 10: entity type with a simple key

```
"Category": {
  "$Kind": "EntityType",
  "$Key": [
    "ID"
  ],
  "ID": {
    "$Type": "Edm.Int32"
  },
  "Name": {
    "$Nullable": true,
    "@Core.IsLanguageDependent": true
  }
}
```

Example 11: entity type with a simple key referencing a property of a complex type

```
"Category": {
  "$Kind": "EntityType",
  "$Key": [
    {
      "EntityInfoID": "Info/ID"
    }
  ],
  "Info": {
    "$Type": "self.EntityInfo"
  },
  "Name": {
    "$Nullable": true
  }
},
"EntityInfo": {
  "$Kind": "ComplexType",
  "ID": {
    "$Type": "Edm.Int32"
  },
  "Created": {
    "$Type": "Edm.DateTimeOffset",
    "$Precision": 0
  }
}
```

Example 12: entity type with a composite key

```
"OrderLine": {
  "$Kind": "EntityType",
  "$Key": [
    "OrderID",
    "LineNumber"
  ],
  "OrderID": {
    "$Type": "Edm.Int32"
  },
  "LineNumber": {
    "$Type": "Edm.Int32"
  }
```

```
}
```

Example 13 (based on example 11): requests to an entity set `Categories` of type `Category` must use the alias

```
GET http://host/service/Categories(EntityInfoID=1)
```

Example 14 (based on example 11): in a query part the value assigned to the name attribute must be used

```
GET http://example.org/OData.svc/Categories?$filter=Info/ID le 100
```

7 Structural Property

A structural property is a property of a structured type that has one of the following types:

- Primitive type
- Complex type
- Enumeration type
- A collection of one of the above

A structural property MUST specify a unique name as well as a type.

The property's name MUST be a simple identifier. It is used when referencing, serializing or deserializing the property. It MUST be unique within the set of structural and navigation properties of the declaring structured type, and MUST NOT match the name of any navigation property in any of its base types. If a structural property with the same name is defined in any of this type's base types, then the property's type MUST be a type derived from the type specified for the property of the base type and constrains this property to be of the specified subtype for instances of this structured type. The name MUST NOT match the name of any structural or navigation property of any of this type's base types for OData 4.0 responses.

Names are case-sensitive, but service authors SHOULD NOT choose names that differ only in case.

Property Object

Structural properties are represented as members of the object representing a structured type. The member name is the property name, the member value is an object.

The property object MAY contain the member `$Kind` with a string value of `Property`. This member SHOULD be omitted to reduce document size.

It MAY contain the member `$Type`, `$Collection`, `$Nullable`, `$MaxLength`, `$Unicode`, `$Precision`, `$Scale`, `$SRID`, and `$DefaultValue`.

It also MAY contain annotations.

Example 15: complex type with two properties `Dimension` and `Length`

```
"Measurement": {
  "$Kind": "ComplexType",
  "Dimension": {
    "$MaxLength": 50,
    "$DefaultValue": "Unspecified"
  },
  "Length": {
    "$Type": "Edm.Decimal",
    "$Precision": 18,
    "$Scale": 2
  }
}
```

7.1 Type

The property's type MUST be a primitive type, complex type, or enumeration type in scope, or a collection of one of these types.

A collection-valued property MAY be annotated with the `Core.Ordered` term, defined in **[OData-VocCore]**, to specify that it supports a stable ordering.

A collection-valued property MAY be annotated with the `Core.PositionalInsert` term, defined in **[OData-VocCore]**, to specify that it supports inserting items into a specific ordinal position.

`$Type` and `$Collection`

For single-valued properties the value of `$Type` is the qualified name of the property's type.

For collection-valued properties the value of `$Type` is the qualified name of the property's item type, and the member `$Collection` MUST be present with the literal value `true`.

Absence of the `$Type` member means the type is `Edm.String`. This member SHOULD be omitted for string properties to reduce document size.

Example 16: property `Units` that can have zero or more strings as its value

```
"Units": {
  "$Collection": true
}
```

7.2 Type Facets

Facets modify or constrain the acceptable values of a property.

For single-valued properties the facets apply to the value of the property. For collection-valued properties the facets apply to the items in the collection.

7.2.1 Nullable

A Boolean value specifying whether the property can have the value `null`.

`$Nullable`

The value of `$Nullable` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

For single-valued properties the value `true` means that the property allows the `null` value.

For collection-valued properties the property value will always be a collection that MAY be empty. In this case `$Nullable` applies to items of the collection and specifies whether the collection MAY contain `null` values.

7.2.2 MaxLength

A positive integer value specifying the maximum length of a binary, stream or string value. For binary or stream values this is the octet length of the binary data, for string values it is the character length (number of code points for Unicode).

If no maximum length is specified, clients SHOULD expect arbitrary length.

`$MaxLength`

The value of `$MaxLength` is a positive integer.

Note: **[OData-CSDLXML]** defines a symbolic value `max` that is only allowed in OData 4.0 responses. This symbolic value is not allowed in CDSL JSON documents at all. Services MAY instead specify the concrete maximum length supported for the type by the service or omit the member entirely.

7.2.3 Precision

For a decimal value: the maximum number of significant decimal digits of the property's value; it MUST be a positive integer.

For a temporal value (datetime-with-timezone-offset, duration, or time-of-day): the number of decimal places allowed in the seconds portion of the value; it MUST be a non-negative integer between zero and twelve.

Note: service authors SHOULD be aware that some clients are unable to support a precision greater than 28 for decimal properties and 7 for temporal properties. Client developers MUST be aware of the potential for data loss when round-tripping values of greater precision. Updating via `PATCH` and exclusively specifying modified properties will reduce the risk for unintended data loss.

Note: duration properties supporting a granularity less than seconds (e.g. minutes, hours, days) can be annotated with term `Measures.DurationGranularity`, see **[OData-VocMeasures]**.

> **`$Precision`**
>
> The value of `$Precision` is a number.
>
> Absence of `$Precision` means arbitrary precision.

Example 17: `Precision` facet applied to the `DateTimeOffset` type

```
"SuggestedTimes": {
  "$Type": Edm.DateTimeOffset",
  "$Collection": true,
  "$Precision": 6
}
```

7.2.4 Scale

A non-negative integer value specifying the maximum number of digits allowed to the right of the decimal point, or one of the symbolic values `floating` or `variable`.

The value `floating` means that the decimal property represents a decimal floating-point number whose number of significant digits is the value of the `Precision` facet. OData 4.0 responses MUST NOT specify the value `floating`.

The value `variable` means that the number of digits to the right of the decimal point can vary from zero to the value of the `Precision` facet.

An integer value means that the number of digits to the right of the decimal point may vary from zero to the value of the `Scale` facet, and the number of digits to the left of the decimal point may vary from one to the value of the `Precision` facet minus the value of the `Scale` facet. If `Precision` is equal to `Scale`, a single zero MUST precede the decimal point.

The value of `Scale` MUST be less than or equal to the value of `Precision`.

Note: if the underlying data store allows negative scale, services may use a `Precision` with the absolute value of the negative scale added to the actual number of significant decimal digits, and client-provided values may have to be rounded before being stored.

> **`$Scale`**
>
> The value of `$Scale` is a number or a string with one of the symbolic values `floating` or `variable`.
>
> Services SHOULD use lower-case values; clients SHOULD accept values in a case-insensitive manner.
>
> Absence of `$Scale` means `variable`.

Example 18: `Precision=3` and `Scale=2`.
Allowed values: 1.23, 0.23, 3.14 and 0.7, not allowed values: 123, 12.3

```
"Amount32": {
  "$Nullable": true,
  "$Type": "Edm.Decimal",
  "$Precision": 3,
  "$Scale": 2
}
```

Example 19: `Precision=2` equals `Scale`.
Allowed values: 0.23, 0.7, not allowed values: 1.23, 1.2

```
"Amount22": {
  "$Nullable": true,
  "$Type": "Edm.Decimal",
  "$Precision": 2,
  "$Scale": 2
}
```

Example 20: `Precision=3` and a variable `Scale`.
Allowed values: 0.123, 1.23, 0.23, 0.7, 123 and 12.3, not allowed values: 12.34, 1234 and 123.4 due to the limited precision.

```
"Amount3v": {
  "$Nullable": true,
  "$Type": "Edm.Decimal",
  "$Precision": 3
}
```

Example 21: `Precision=7` and a floating `Scale`.
Allowed values: -1.234567e3, 1e-101, 9.999999e96, not allowed values: 1e-102 and 1e97 due to the limited precision.

```
"Amount7f": {
  "$Nullable": true,
  "$Type": "Edm.Decimal",
  "$Precision": 7,
  "$Scale": "floating"
}
```

7.2.5 Unicode

For a string property the `Unicode` facet indicates whether the property might contain and accept string values with Unicode characters (code points) beyond the ASCII character set. The value `false` indicates that the property will only contain and accept string values with characters limited to the ASCII character set.

If no value is specified, the `Unicode` facet defaults to `true`.

> **`$Unicode`**
>
> The value of `$Unicode` is one of the Boolean literals `true` or `false`. Absence of the member means `true`.

7.2.6 SRID

For a geometry or geography property the `SRID` facet identifies which spatial reference system is applied to values of the property on type instances.

The value of the `SRID` facet MUST be a non-negative integer or the special value `variable`. If no value is specified, the facet defaults to `0` for `Geometry` types or `4326` for `Geography` types.

The valid values of the `SRID` facet and their meanings are as defined by the European Petroleum Survey Group **[EPSG]**.

> **`$SRID`**
>
> The value of `$SRID` is a string containing a number or the symbolic value `variable`.

7.2.7 Default Value

A primitive or enumeration property MAY define a default value that is used if the property is not explicitly represented in an annotation or the body of a request or response.

If no value is specified, the client SHOULD NOT assume a default value.

> **`$DefaultValue`**
>
> The value of `$DefaultValue` is the type-specific JSON representation of the default value of the property, see **[OData-JSON]**. For properties of type `Edm.Decimal` and `Edm.Int64` the representation depends on the media type parameter `IEEE754Compatible`.

8 Navigation Property

A navigation property allows navigation to related entities. It MUST specify a unique name as well as a type.

The navigation property's name MUST be a simple identifier. It is used when referencing, serializing or deserializing the navigation property. It MUST be unique within the set of structural and navigation properties of the declaring structured type, and MUST NOT match the name of any structural property in any of its base types. If a navigation property with the same name is defined in any of this type's base types, then the navigation property's type MUST be a type derived from the type specified for the navigation property of the base type, and constrains this navigation property to be of the specified subtype for instances of this structured type. The name MUST NOT match the name of any structural or navigation property of any of this type's base types for OData 4.0 responses.

Names are case-sensitive, but service authors SHOULD NOT choose names that differ only in case.

Navigation Property Object

Navigation properties are represented as members of the object representing a structured type. The member name is the property name, the member value is an object.

The navigation property object MUST contain the member `$Kind` with a string value of `NavigationProperty`.

It MUST contain the member `$Type`, and it MAY contain the members `$Collection`, `$Nullable`, `$Partner`, `$ContainsTarget`, `$ReferentialConstraint`, and `$OnDelete`.

It also MAY contain annotations.

Example 22: the `Product` entity type has a navigation property to a `Category`, which has a navigation link back to one or more products

```
"Product": {
  "$Kind": "EntityType",
  …
  "Category": {
    "$Kind": "NavigationProperty",
    "$Type": "self.Category",
    "$Partner": "Products",
    "$ReferentialConstraint": {
      "CategoryID": "ID"
    }
  },
  "Supplier": {
    "$Kind": "NavigationProperty",
    "$Type": "self.Supplier"
  }
},
"Category": {
  "$Kind": "EntityType",
  …
  "Products": {
    "$Kind": "NavigationProperty",
    "$Collection": true,
    "$Type": "self.Product",
    "$Partner": "Category",
    "$OnDelete": "Cascade",
    "$OnDelete@Core.Description": "Delete all related entities"
  }
}
```

8.1 Navigation Property Type

The navigation property's type MUST be an entity type in scope, the abstract type `Edm.EntityType`, or a collection of one of these types.

If the type is a collection, an arbitrary number of entities can be related. Otherwise there is at most one related entity.

The related entities MUST be of the specified entity type or one of its subtypes.

For a collection-valued containment navigation property the specified entity type MUST have a key defined.

A collection-valued navigation property MAY be annotated with the `Core.Ordered` term, defined in **[OData-VocCore]**, to specify that it supports a stable ordering.

A collection-valued navigation property MAY be annotated with the `Core.PositionalInsert` term, defined in **[OData-VocCore]**, to specify that it supports inserting items into a specific ordinal position.

> ### `$Type` and `$Collection`
>
> For single-valued navigation properties the value of `$Type` is the qualified name of the navigation property's type.
>
> For collection-valued navigation properties the value of `$Type` is the qualified name of the navigation property's item type, and the member `$Collection` MUST be present with the literal value `true`.

8.2 Nullable Navigation Property

A Boolean value specifying whether the declaring type MAY have no related entity. If false, instances of the declaring structured type MUST always have a related entity.

Nullable MUST NOT be specified for a collection-valued navigation property, a collection is allowed to have zero items.

> ### `$Nullable`
>
> The value of `$Nullable` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

8.3 Partner Navigation Property

A navigation property of an entity type MAY specify a partner navigation property. Navigation properties of complex types MUST NOT specify a partner.

If specified, the partner navigation property is identified by a path relative to the entity type specified as the type of the navigation property. This path MUST lead to a navigation property defined on that type or a derived type. The path MAY traverse complex types, including derived complex types, but MUST NOT traverse any navigation properties. The type of the partner navigation property MUST be the declaring entity type of the current navigation property or one of its parent entity types.

If the partner navigation property is single-valued, it MUST lead back to the source entity from all related entities. If the partner navigation property is collection-valued, the source entity MUST be part of that collection.

If no partner navigation property is specified, no assumptions can be made as to whether one of the navigation properties on the target type will lead back to the source entity.

If a partner navigation property is specified, this partner navigation property MUST either specify the current navigation property as its partner to define a bi-directional relationship or it MUST NOT specify a partner navigation property. The latter can occur if the partner navigation property is defined on a complex type, or if the current navigation property is defined on a type derived from the type of the partner navigation property.

$Partner

The value of `$Partner` is a string containing the path to the partner navigation property.

8.4 Containment Navigation Property

A navigation property MAY indicate that instances of its declaring structured type contain the targets of the navigation property, in which case the navigation property is called a *containment navigation property*.

Containment navigation properties define an implicit entity set for each instance of its declaring structured type. This implicit entity set is identified by the read URL of the navigation property for that structured type instance.

Instances of the structured type that declares the navigation property, either directly or indirectly via a property of complex type, contain the entities referenced by the containment navigation property. The canonical URL for contained entities is the canonical URL of the containing instance, followed by the path segment of the navigation property and the key of the contained entity, see **[OData-URL]**.

Entity types used in collection-valued containment navigation properties MUST have a key defined.

For items of an ordered collection of complex types (those annotated with the `Core.Ordered` term defined in **[OData-VocCore]**), the canonical URL of the item is the canonical URL of the collection appended with a segment containing the zero-based ordinal of the item. Items within in an unordered collection of complex types do not have a canonical URL. Services that support unordered collections of complex types declaring a containment navigation property, either directly or indirectly via a property of complex type, MUST specify the URL for the navigation link within a payload representing that item, according to format-specific rules.

OData 4.0 responses MUST NOT specify a complex type declaring a containment navigation property as the type of a collection-valued property.

An entity cannot be referenced by more than one containment relationship, and cannot both belong to an entity set declared within the entity container and be referenced by a containment relationship.

Containment navigation properties MUST NOT be specified as the last path segment in the path of a navigation property binding.

When a containment navigation property navigates between entity types in the same inheritance hierarchy, the containment is called *recursive*.

Containment navigation properties MAY specify a partner navigation property. If the containment is recursive, the relationship defines a tree, thus the partner navigation property MUST be nullable (for the root of the tree) and single-valued (for the parent of a non-root entity). If the containment is not recursive, the partner navigation property MUST NOT be nullable.

An entity type inheritance chain MUST NOT contain more than one navigation property with a partner navigation property that is a containment navigation property.

Note: without a partner navigation property, there is no reliable way for a client to determine which entity contains a given contained entity. This may lead to problems for clients if the contained entity can also be reached via a non-containment navigation path.

$ContainsTarget

The value of `$ContainsTarget` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

8.5 Referential Constraint

A single-valued navigation property MAY define one or more referential constraints. A referential constraint asserts that the *dependent property* (the property defined on the structured type declaring the navigation property) MUST have the same value as the *principal property* (the referenced property declared on the entity type that is the target of the navigation).

The type of the dependent property MUST match the type of the principal property, or both types MUST be complex types.

If the principle property references an entity, then the dependent property must reference the same entity.

If the principle property's value is a complex type instance, then the dependent property's value must be a complex type instance with the same properties, each with the same values.

If the navigation property on which the referential constraint is defined is nullable, or the principal property is nullable, then the dependent property MUST also be nullable. If both the navigation property and the principal property are not nullable, then the dependent property MUST NOT be nullable.

$ReferentialConstraint

The value of `$ReferentialConstraint` is an object with one member per referential constraint. The member name is the path to the dependent property, this path is relative to the structured type declaring the navigation property. The member value is a string containing the path to the principal property, this path is relative to the entity type that is the target of the navigation property.

It also MAY contain annotations. These are prefixed with the path of the dependent property of the annotated referential constraint.

Example 23: the category must exist for a product in that category to exist. The `CategoryID` of the product is identical to the `ID` of the category, and the `CategoryKind` property of the product is identical to the `Kind` property of the category.

```
"Product": {
  "$Kind": "EntityType",
  …
  "CategoryID": {},
  "CategoryKind": {},
  "Category": {
    "$Kind": "NavigationProperty",
    "$Type": "self.Category",
    "$Partner": "Products",
    "$ReferentialConstraint": {
      "CategoryID": "ID",
      "CategoryKind": "Kind"
      "CategoryKind@Core.Description": "Referential Constraint to non-key
property"
    }
  }
},
"Category": {
  "$Kind": "EntityType",
  "$Key": [
    "ID"
  ],
  "ID": {},
  "Kind": {
    "$Nullable": true
  },
  …
}
```

8.6 On-Delete Action

A navigation property MAY define an on-delete action that describes the action the service will take on related entities when the entity on which the navigation property is defined is deleted.

The action can have one of the following values:

- `Cascade`, meaning the related entities will be deleted if the source entity is deleted,
- `None`, meaning a `DELETE` request on a source entity with related entities will fail,

- `SetNull`, meaning all properties of related entities that are tied to properties of the source entity via a referential constraint and that do not participate in other referential constraints will be set to null,
- `SetDefault`, meaning all properties of related entities that are tied to properties of the source entity via a referential constraint and that do not participate in other referential constraints will be set to their default value.

If no on-delete action is specified, the action taken by the service is not predictable by the client and could vary per entity.

`$OnDelete`

The value of `$OnDelete` is a string with one of the values `Cascade`, `None`, `SetNull`, or `SetDefault`.

Annotations for `$OnDelete` are prefixed with `$OnDelete`.

Example 24: deletion of a category implies deletion of the related products in that category

```
"Category": {
  "$Kind": "EntityType",
  …
  "Products": {
    "$Kind": "NavigationProperty",
    "$Collection": true,
    "$Type": "self.Product",
    "$Partner": "Category",
    "$OnDelete": "Cascade",
    "$OnDelete@Core.Description": "Delete all products in this category"
  }
}
```

9 Complex Type

Complex types are keyless nominal structured types. The lack of a key means that instances of complex types cannot be referenced, created, updated or deleted independently of an entity type. Complex types allow entity models to group properties into common structures.

The complex type's name is a simple identifier that MUST be unique within its schema.

A complex type can define two types of properties. A structural property is a named reference to a primitive, complex, or enumeration type, or a collection of primitive, complex, or enumeration types. A navigation property is a named reference to an entity type or a collection of entity types.

All properties MUST have a unique name within a complex type. Properties MUST NOT have the same name as the declaring complex type. They MAY have the same name as one of the direct or indirect base types or derived types.

Complex Type Object

A complex type is represented as a member of the schema object whose name is the unqualified name of the complex type and whose value is an object.

The complex type object MUST contain the member `$Kind` with a string value of `ComplexType`. It MAY contain the members `$BaseType`, `$Abstract`, and `$OpenType`. It also MAY contain members representing structural properties and navigation properties as well as annotations.

Example 25: a complex type used by two entity types

```
"Dimensions": {
  "$Kind": "ComplexType",
  "Height": {
    "$Type": "Edm.Decimal",
    "$Scale": 0
  },
  "Weight": {
    "$Type": "Edm.Decimal",
    "$Scale": 0
  },
  "Length": {
    "$Type": "Edm.Decimal",
    "$Scale": 0
  }
},
"Product": {
  …
  "ProductDimensions": {
    "$Nullable": true,
    "$Type": "self.Dimensions"
  },
  "ShippingDimensions": {
    "$Nullable": true,
    "$Type": "self.Dimensions"
  }
},
"ShipmentBox": {
  …
  "Dimensions": {
    "$Nullable": true,
    "$Type": "self.Dimensions"
  }
}
```

9.1 Derived Complex Type

A complex type can inherit from another complex type by specifying it as its base type.

A complex type inherits the structural and navigation properties of its base type.

A complex type MUST NOT introduce an inheritance cycle by specifying a base type.

The rules for annotations of derived complex types are described in section 14.2.

> **`$BaseType`**
>
> The value of `$BaseType` is the qualified name of the base type.

9.2 Abstract Complex Type

A complex type MAY indicate that it is abstract and cannot have instances.

> **`$Abstract`**
>
> The value of `$Abstract` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

9.3 Open Complex Type

A complex type MAY indicate that it is open and allows clients to add properties dynamically to instances of the type by specifying uniquely named property values in the payload used to insert or update an instance of the type.

A complex type derived from an open complex type MUST indicate that it is also open.

Note: structural and navigation properties MAY be returned by the service on instances of any structured type, whether or not the type is marked as open. Clients MUST always be prepared to deal with additional properties on instances of any structured type, see **[OData-Protocol]**.

> **`$OpenType`**
>
> The value of `$OpenType` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

10 Enumeration Type

Enumeration types are nominal types that represent a non-empty series of related values. Enumeration types expose these related values as members of the enumeration.

The enumeration type's name is a simple identifier that MUST be unique within its schema.

Although enumeration types have an underlying numeric value, the preferred representation for an enumeration value is the member name. Discrete sets of numeric values should be represented as numeric values annotated with the `AllowedValues` annotation defined in **[OData-VocCore]**.

Enumeration types marked as flags allow values that consist of more than one enumeration member at a time.

Enumeration Type Object

An enumeration type is represented as a member of the schema object whose name is the unqualified name of the enumeration type and whose value is an object.

The enumeration type object MUST contain the member `$Kind` with a string value of `EnumType`.

It MAY contain the members `$UnderlyingType` and `$IsFlags`.

The enumeration type object MUST contain members representing the enumeration type members.

The enumeration type object MAY contain annotations.

Example 26: a simple flags-enabled enumeration

```
"FileAccess": {
  "$Kind": "EnumType",
  "$UnderlyingType": "Edm.Int32",
  "$IsFlags": true,
  "Read": 1,
  "Write": 2,
  "Create": 4,
  "Delete": 8
}
```

10.1 Underlying Integer Type

An enumeration type MAY specify one of `Edm.Byte`, `Edm.SByte`, `Edm.Int16`, `Edm.Int32`, or `Edm.Int64` as its underlying type.

If not explicitly specified, `Edm.Int32` is used as the underlying type.

$UnderlyingType

The value of `$UnderlyingType` is the qualified name of the underlying type.

10.2 Flags Enumeration Type

An enumeration type MAY indicate that the enumeration type allows multiple members to be selected simultaneously.

If not explicitly specified, only one enumeration type member MAY be selected simultaneously.

$IsFlags

The value of `$IsFlags` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

Example 27: pattern values can be combined, and some combined values have explicit names

```
"Pattern": {
  "$Kind": "EnumType",
  "$UnderlyingType": "Edm.Int32",
  "$IsFlags": true,
  "Plain": 0,
  "Red": 1,
  "Blue": 2,
  "Yellow": 4,
  "Solid": 8,
  "Striped": 16,
  "SolidRed": 9,
  "SolidBlue": 10,
  "SolidYellow": 12,
  "RedBlueStriped": 19,
  "RedYellowStriped": 21,
  "BlueYellowStriped": 22
}
```

10.3 Enumeration Type Member

Enumeration type values consist of discrete members.

Each member is identified by its name, a simple identifier that MUST be unique within the enumeration type. Names are case-sensitive, but service authors SHOULD NOT choose names that differ only in case.

Each member MUST specify an associated numeric value that MUST be a valid value for the underlying type of the enumeration type.

Enumeration types can have multiple members with the same value. Members with the same numeric value compare as equal, and members with the same numeric value can be used interchangeably.

Enumeration members are sorted by their numeric value.

For flag enumeration types the combined numeric value of simultaneously selected members is the bitwise OR of the discrete numeric member values.

Enumeration Member Object

Enumeration type members are represented as JSON object members, where the object member name is the enumeration member name and the object member value is the enumeration member value.

For members of flags enumeration types a combined enumeration member value is equivalent to the bitwise OR of the discrete values.

Annotations for enumeration members are prefixed with the enumeration member name.

Example 28: `FirstClass` has a value of `0`, `TwoDay` a value of 1, and `Overnight` a value of 2.

```
"ShippingMethod": {
  "$Kind": "EnumType",
  "FirstClass": 0,
  "FirstClass@Core.Description": "Shipped with highest priority",
  "TwoDay": 1,
  "TwoDay@Core.Description": "Shipped within two days",
  "Overnight": 2,
  "Overnight@Core.Description": "Shipped overnight",
  "@Core.Description": "Method of shipping"
}
```

11 Type Definition

A type definition defines a specialization of one of the primitive types or of the built-in abstract type `Edm.PrimitiveType`.

The type definition's name is a simple identifier that MUST be unique within its schema.

Type definitions can be used wherever a primitive type is used (other than as the underlying type in a new type definition) and are type-comparable with their underlying types and any type definitions defined using the same underlying type.

It is up to the definition of a term to specify whether and how annotations with this term propagate to places where the annotated type definition is used, and whether they can be overridden.

Type Definition Object

A type definition is represented as a member of the schema object whose name is the unqualified name of the type definition and whose value is an object.

The type definition object MUST contain the member `$Kind` with a string value of `TypeDefinition` and the member `$UnderlyingType`. It MAY contain the members `$MaxLength`, `$Unicode`, `$Precision`, `$Scale`, and `$SRID`, and it MAY contain annotations.

Example 29:

```
"Length": {
  "$Kind": "TypeDefinition",
  "$UnderlyingType": "Edm.Int32",
  "@Measures.Unit": "Centimeters"
},
"Weight": {
  "$Kind": "TypeDefinition",
  "$UnderlyingType": "Edm.Int32",
  "@Measures.Unit": "Kilograms"
},
"Size": {
  "$Kind": "ComplexType",
  "Height": {
    "$Nullable": true,
    "$Type": "self.Length"
  },
  "Weight": {
    "$Nullable": true,
    "$Type": "self.Weight"
  }
}
```

11.1 Underlying Primitive Type

The underlying type of a type definition MUST be a primitive type that MUST NOT be another type definition.

$UnderlyingType

The value of `$UnderlyingType` is the qualified name of the underlying type.

The type definition MAY specify facets applicable to the underlying type. Possible facets are: `$MaxLength`, `$Unicode`, `$Precision`, `$Scale`, or `$SRID`.

Additional facets appropriate for the underlying type MAY be specified when the type definition is used but the facets specified in the type definition MUST NOT be re-specified.

For a type definition with underlying type `Edm.PrimitiveType` no facets are applicable, neither in the definition itself nor when the type definition is used, and these should be ignored by the client.

Where type definitions are used, the type definition is returned in place of the primitive type wherever the type is specified in a response.

12 Action and Function

12.1 Action

Actions are service-defined operations that MAY have observable side effects and MAY return a single instance or a collection of instances of any type.

The action's name is a simple identifier that MUST be unique within its schema.

Actions cannot be composed with additional path segments.

An action MAY specify a return type that MUST be a primitive, entity or complex type, or a collection of primitive, entity or complex types in scope.

An action MAY define parameters used during the execution of the action.

12.2 Action Overloads

Bound actions support overloading (multiple actions having the same name within the same schema) by binding parameter type. The combination of action name and the binding parameter type MUST be unique within a schema.

Unbound actions do not support overloads. The names of all unbound actions MUST be unique within a schema.

An unbound action MAY have the same name as a bound action.

Action Overload Object

An action is represented as a member of the schema object whose name is the unqualified name of the action and whose value is an array. The array contains one object per action overload.

The action overload object MUST contain the member `$Kind` with a string value of `Action`.

It MAY contain the members `$IsBound`, `$EntitySetPath`, `$Parameter`, and `$ReturnType`, and it MAY contain annotations.

12.3 Function

Functions are service-defined operations that MUST NOT have observable side effects and MUST return a single instance or a collection of instances of any type.

The function's name is a simple identifier that MUST be unique within its schema.

Functions MAY be composable.

The function MUST specify a return type which MUST be a primitive, entity or complex type, or a collection of primitive, entity or complex types in scope.

A function MAY define parameters used during the execution of the function.

12.4 Function Overloads

Bound functions support overloading (multiple functions having the same name within the same schema) subject to the following rules:

- The combination of function name, binding parameter type, and unordered set of non-binding parameter names MUST be unique within a schema.
- The combination of function name, binding parameter type, and ordered set of parameter types MUST be unique within a schema.
- All bound functions with the same function name and binding parameter type within a schema MUST specify the same return type.

Unbound functions support overloading subject to the following rules:

- The combination of function name and unordered set of parameter names MUST be unique within a schema.
- The combination of function name and ordered set of parameter types MUST be unique within a schema.
- All unbound functions with the same function name within a schema MUST specify the same return type.

An unbound function MAY have the same name as a bound function.

Note that type definitions can be used to disambiguate overloads for both bound and unbound functions, even if they specify the same underlying type.

> ### Function Overload Object
>
> A function is represented as a member of the schema object whose name is the unqualified name of the function and whose value is an array. The array contains one object per function overload.
>
> The function overload object MUST contain the member `$Kind` with a string value of `Function`.
>
> It MUST contain the member `$ReturnType`, and it MAY contain the members `$IsBound`, `$EntitySetPath`, and `$Parameter`, and it MAY contain annotations.

12.5 Bound or Unbound Action or Function Overloads

An action or function overload MAY indicate that it is bound. If not explicitly indicated, it is unbound.

Bound actions or functions are invoked on resources matching the type of the binding parameter. The binding parameter can be of any type, and it MAY be nullable.

Unbound actions are invoked from the entity container through an action import.

Unbound functions are invoked as static functions within a filter or orderby expression, or from the entity container through a function import.

> ### `$IsBound`
>
> The value of `$IsBound` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

12.6 Entity Set Path

Bound actions and functions that return an entity or a collection of entities MAY specify an entity set path if the entity set of the returned entities depends on the entity set of the binding parameter value.

The entity set path consists of a series of segments joined together with forward slashes.

The first segment of the entity set path MUST be the name of the binding parameter. The remaining segments of the entity set path MUST represent navigation segments or type casts.

A navigation segment names the simple identifier of the navigation property to be traversed. A type-cast segment names the qualified name of the entity type that should be returned from the type cast.

> ### `$EntitySetPath`
>
> The value of `$EntitySetPath` is a string containing the entity set path.

12.7 Composable Function

A function MAY indicate that it is composable. If not explicitly indicated, it is not composable.

A composable function can be invoked with additional path segments or key predicates appended to the resource path that identifies the composable function, and with system query options as appropriate for the type returned by the composable function.

$IsComposable

The value of `$IsComposable` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

12.8 Return Type

The return type of an action or function overload MAY be any type in scope, or a collection of any type in scope.

The facets `Nullable`, `MaxLength`, `Precision`, `Scale`, and `SRID` can be used as appropriate to specify value restrictions of the return type, as well as the `Unicode` facet for 4.01 and greater payloads.

For a single-valued return type the facets apply to the returned value. For a collection-valued return type the facets apply to the items in the returned collection.

$ReturnType

The value of `$ReturnType` is an object. It MAY contain the members `$Type`, `$Collection`, `$Nullable`, `$MaxLength`, `$Unicode`, `$Precision`, `$Scale`, and `$SRID`.

It also MAY contain annotations.

$Type and $Collection

For single-valued return types the value of `$Type` is the qualified name of the returned type.

For collection-valued return types the value of `$Type` is the qualified name of the returned item type, and the member `$Collection` MUST be present with the literal value `true`.

Absence of the `$Type` member means the type is `Edm.String`.

$Nullable

The value of `$Nullable` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

If the return type is a collection of entity types, the `$Nullable` member has no meaning and MUST NOT be specified.

For other collection-valued return types the result will always be a collection that MAY be empty. In this case `$Nullable` applies to items of the collection and specifies whether the collection MAY contain `null` values.

For single-valued return types the value `true` means that the action or function MAY return a single `null` value. The value `false` means that the action or function will never return a `null` value and instead will fail with an error response if it cannot compute a result.

12.9 Parameter

An action or function overload MAY specify parameters.

A bound action or function overload MUST specify at least one parameter; the first parameter is its binding parameter. The order of parameters MUST NOT change unless the schema version changes.

Each parameter MUST have a name that is a simple identifier. The parameter name MUST be unique within the action or function overload.

The parameter MUST specify a type. It MAY be any type in scope, or a collection of any type in scope.

The facets `MaxLength`, `Precision`, `Scale`, or `SRID` can be used as appropriate to specify value restrictions of the parameter, as well as the `Unicode` facet for 4.01 and greater payloads.

For single-valued parameters the facets apply to the parameter value. If the parameter value is a collection, the facets apply to the items in the collection.

`$Parameter`

The value of `$Parameter` is an array. The array contains one object per parameter.

Parameter Object

A parameter object MUST contain the member `$Name`, and it MAY contain the members `$Type`, `$Collection`, `$Nullable`, `$MaxLength`, `$Unicode`, `$Precision`, `$Scale`, and `$SRID`.

Parameter objects MAY also contain annotations.

`$Name`

The value of `$Name` is a string containing the parameter name.

`$Type` and `$Collection`

For single-valued parameters the value of `$Type` is the qualified name of the accepted type.

For collection-valued parameters the value of `$Type` is the qualified name of the accepted item type, and the member `$Collection` MUST be present with the literal value `true`.

Absence of the `$Type` member means the type is `Edm.String`.

`$Nullable`

The value of `$Nullable` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

For single-valued parameters the value `true` means that the parameter accepts a `null` value.

For collection-valued parameters the parameter value will always be a collection that MAY be empty. In this case `$Nullable` applies to items of the collection and specifies whether the collection MAY contain `null` values.

Example 30: a function returning the top-selling products for a given year. In this case the year must be specified as a parameter of the function with the `edm:Parameter` element.

```
"TopSellingProducts": [
  {
    "$Kind": "Function",
    "$Parameter": [
      {
        "$Name": "Year",
        "$Nullable": true,
        "$Type": "Edm.Decimal",
        "$Precision": 4,
        "$Scale": 0
      }
    ],
    "$ReturnType": {
      "$Collection": true,
      "$Type": "self.Product"
    }
  }
]
```

13 Entity Container

Each metadata document used to describe an OData service MUST define exactly one entity container.

The entity container's name is a simple identifier that MUST be unique within its schema.

Entity containers define the entity sets, singletons, function and action imports exposed by the service.

Entity set, singleton, action import, and function import names MUST be unique within an entity container.

An *entity set* allows access to entity type instances. Simple entity models frequently have one entity set per entity type.

Example 31: one entity set per entity type

```
"Products": {
  "$Collection": true,
  "$Type": "self.Product"
},
"Categories": {
  "$Collection": true,
  "$Type": "self.Category"
}
```

Other entity models may expose multiple entity sets per type.

Example 32: three entity sets referring to the two entity types

```
"StandardCustomers": {
  "$Collection": true,
  "$Type": "self.Customer",
  "$NavigationPropertyBinding": {
    "Orders": "Orders"
  }
},
"PreferredCustomers": {
  "$Collection": true,
  "$Type": "self.Customer",
  "$NavigationPropertyBinding": {
    "Orders": "Orders"
  }
},
"Orders": {
  "$Collection": true,
  "$Type": "self.Order"
}
```

There are separate entity sets for standard customers and preferred customers, but only one entity set for orders. The entity sets for standard customers and preferred customers both have navigation property bindings to the orders entity set, but the orders entity set does not have a navigation property binding for the Customer navigation property, since it could lead to either set of customers.

An entity set can expose instances of the specified entity type as well as any entity type inherited from the specified entity type.

A *singleton* allows addressing a single entity directly from the entity container without having to know its key, and without requiring an entity set.

A *function import* or an *action import* is used to expose a function or action defined in an entity model as a top level resource.

Entity Container Object

An entity container is represented as a member of the schema object whose name is the unqualified name of the entity container and whose value is an object.

The entity container object MUST contain the member `$Kind` with a string value of `EntityContainer`.

The entity container object MAY contain the member `$Extends`, members representing entity sets, singletons, action imports, and function imports, as well as annotations.

Example 33: An entity container aggregates entity sets, singletons, action imports, and function imports.

```
"DemoService": {
  "$Kind": "EntityContainer",
  "Products": {
    "$Collection": true,
    "$Type": "self.Product",
    "$NavigationPropertyBinding": {
      "Category": "Categories",
      "Supplier": "Suppliers"
    },
    "@UI.DisplayName": "Product Catalog"
  },
  "Categories": {
    "$Collection": true,
    "$Type": "self.Category",
    "$NavigationPropertyBinding": {
      "Products": "Products"
    }
  },
  "Suppliers": {
    "$Collection": true,
    "$Type": "self.Supplier",
    "$NavigationPropertyBinding": {
      "Products": "Products"
    },
    "@UI.DisplayName": "Supplier Directory"
  },
  "MainSupplier": {
    "$Type": "self.Supplier"
  },
  "LeaveRequestApproval": {
    "$Action": "self.Approval"
  },
  "ProductsByRating": {
    "$EntitySet": "Products",
    "$Function": "self.ProductsByRating"
  }
}
```

13.1 Extending an Entity Container

An entity container MAY specify that it extends another entity container in scope. All children of the “base” entity container are added to the “extending” entity container.

If the “extending” entity container defines an entity set with the same name as defined in any of its “base” containers, then the entity set’s type MUST specify an entity type derived from the entity type specified for the identically named entity set in the “base” container. The same holds for singletons. Action imports and function imports cannot be redefined, nor can the “extending” container define a child with the same name as a child of a different kind in a “base” container.

Note: services should not introduce cycles by extending entity containers. Clients should be prepared to process cycles introduced by extending entity containers.

`$Extends`

The value of `$Extends` is the qualified name of the entity container to be extended.

Example 34: the entity container `Extending` will contain all child elements that it defines itself, plus all child elements of the `Base` entity container located in `SomeOtherSchema`

```
"Extending": {
  "$Kind": "EntityContainer",
  "$Extends": "Some.Other.Schema.Base",
   …
}
```

13.2 Entity Set

Entity sets are top-level collection-valued resources.

An entity set is identified by its name, a simple identifier that MUST be unique within its entity container.

An entity set MUST specify a type that MUST be an entity type in scope.

An entity set MUST contain only instances of its specified entity type or its subtypes. The entity type MAY be abstract but MUST have a key defined.

An entity set MAY indicate whether it is included in the service document. If not explicitly indicated, it is included.

Entity sets that cannot be queried without specifying additional query options SHOULD NOT be included in the service document.

Entity Set Object

An entity set is represented as a member of the entity container object whose name is the name of the entity set and whose value is an object.

The entity set object MUST contain the members `$Collection` and `$Type`.

It MAY contain the members `$IncludeInServiceDocument` and `$NavigationPropertyBinding` as well as annotations.

`$Collection`

The value of `$Collection` is the Booelan value `true`.

`$Type`

The value of `$Type` is the qualified name of an entity type.

`$IncludeInServiceDocument`

The value of `$IncludeInServiceDocument` is one of the Boolean literals `true` or `false`. Absence of the member means `true`.

13.3 Singleton

Singletons are top-level single-valued resources.

A singleton is identified by its name, a simple identifier that MUST be unique within its entity container.

A singleton MUST specify a type that MUST be an entity type in scope.

A singleton MUST reference an instance its entity type.

Singleton Object

A singleton is represented as a member of the entity container object whose name is the name of the singleton and whose value is an object.

The singleton object MUST contain the member `$Type` and it MAY contain the member `$Nullable`.

It MAY contain the member `$NavigationPropertyBinding` as well as annotations.

`$Type`

The value of `$Type` is the qualified name of an entity type.

`$Nullable`

The value of `$Nullable` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.In OData 4.0 responses this member MUST NOT be specified.

13.4 Navigation Property Binding

If the entity type of an entity set or singleton declares navigation properties, a navigation property binding allows describing which entity set or singleton will contain the related entities.

An entity set or a singleton SHOULD specify a navigation property binding for each navigation property of its entity type, including navigation properties defined on complex typed properties or derived types.

If omitted, clients MUST assume that the target entity set or singleton can vary per related entity.

13.4.1 Navigation Property Path Binding

A navigation property binding MUST specify a path to a navigation property of the entity set's or singleton's declared entity type, or a navigation property reached through a chain of type casts, complex properties, or containment navigation properties. If the navigation property is defined on a subtype, the path MUST contain the qualified name of the subtype, followed by a forward slash, followed by the navigation property name. If the navigation property is defined on a complex type used in the definition of the entity set's entity type, the path MUST contain a forward-slash separated list of complex property names and qualified type names that describe the path leading to the navigation property.

The path can traverse one or more containment navigation properties, but the last navigation property segment MUST be a non-containment navigation property and there MUST NOT be any non-containment navigation properties prior to the final navigation property segment.

If the path traverses collection-valued complex properties or collection-valued containment navigation properties, the binding applies to all items of these collections.

If the path contains a recursive sub-path (i.e. a path leading back to the same structured type, the binding applies recursively to any positive number of cycles through that sub-path.

OData 4.01 services MAY have a type-cast segment as the last path segment, allowing to bind instances of different sub-types to different targets.

The same navigation property path MUST NOT be specified in more than one navigation property binding; navigation property bindings are only used when all related entities are known to come from a single entity set. Note that it is possible to have navigation property bindings for paths that differ only in a type-cast segment, allowing to bind instances of different sub-types to different targets. If paths differ only in type-cast segments, the most specific path applies.

13.4.2 Binding Target

A navigation property binding MUST specify a target via a simple identifier or target path. It specifies the entity set, singleton, or containment navigation property that contains the related entities.

If the target is a simple identifier, it MUST resolve to an entity set or singleton defined in the same entity container.

If the target is a target path, it MUST resolve to an entity set, singleton, or direct or indirect containment navigation property of a singleton in scope. The path can traverse single-valued containment navigation properties or single-valued complex properties before ending in a containment navigation property, and there MUST NOT be any non-containment navigation properties prior to the final segment.

$NavigationPropertyBinding

The value of `$NavigationPropertyBinding` is an object. It consists of members whose name is the navigation property binding path and whose value is a string containing the navigation property binding target. If the target is in the same entity container, the target MUST NOT be prefixed with the qualified entity container name.

Example 35: for an entity set in the same container as the enclosing entity set `Categories`

```
"Categories": {
  "$Collection": true,
  "$Type": "self.Category",
  "$NavigationPropertyBinding": {
    "Products": "SomeSet"
  }
}
```

Example 36: for an entity set in any container in scope

```
"Categories": {
  "$Collection": true,
  "$Type": "self.Category",
  "$NavigationPropertyBinding": {
    "Products": "SomeModel.SomeContainer/SomeSet"
  }
}
```

Example 37: binding `Supplier` on `Products` contained within `Categories` – binding applies to all suppliers of all products of all categories

```
"Categories": {
  "$Collection": true,
  "$Type": "self.Category",
  "$NavigationPropertyBinding": {
    "Products/Supplier": "Suppliers"
  }
}
```

13.5 Action Import

Action imports sets are top-level resources that are never included in the service document.

An action import is identified by its name, a simple identifier that MUST be unique within its entity container.

An action import MUST specify the name of an unbound action in scope.

If the imported action returns an entity or a collection of entities, a simple identifier or target path value MAY be specified to identify the entity set that contains the returned entities. If a simple identifier is specified, it MUST resolve to an entity set defined in the same entity container. If a target path is specified, it MUST resolve to an entity set in scope.

Action Import Object

An action import is represented as a member of the entity container object whose name is the name of the action import and whose value is an object.

The action import object MUST contain the member `$Action`.

It MAY contain the member `$EntitySet`.

It MAY also contain annotations.

$Action

The value of `$Action` is a string containing the qualified name of an unbound action.

$EntitySet

The value of `$EntitySet` is a string containing either the unqualified name of an entity set in the same entity container or a path to an entity set in a different entity container.

13.6 Function Import

Function imports sets are top-level resources.

A function import is identified by its name, a simple identifier that MUST be unique within its entity container.

A function import MUST specify the name of an unbound function in scope. All unbound overloads of the imported function can be invoked from the entity container.

If the imported function returns an entity or a collection of entities, a simple identifier or target path value MAY be specified to identify the entity set that contains the returned entities. If a simple identifier is specified, it MUST resolve to an entity set defined in the same entity container. If a target path is specified, it MUST resolve to an entity set in scope.

A function import for a parameterless function MAY indicate whether it is included in the service document. If not explicitly indicated, it is not included.

Function Import Object

A function import is represented as a member of the entity container object whose name is the name of the function import and whose value is an object.

The function import object MUST contain the member `$Function`.

It MAY contain the members `$EntitySet` and `$IncludeInServiceDocument`.

It MAY also contain annotations.

$Function

The value of `$Function` is a string containing the qualified name of an unbound function.

$EntitySet

The value of `$EntitySet` is a string containing either the unqualified name of an entity set in the same entity container or a path to an entity set in a different entity container.

$IncludeInServiceDocument

The value of `$IncludeInServiceDocument` is one of the Boolean literals `true` or `false`. Absence of the member means `false`.

14 Vocabulary and Annotation

Vocabularies and annotations provide the ability to annotate metadata as well as instance data, and define a powerful extensibility point for OData. An *annotation* applies a *term* to a model element and defines how to calculate a value for the applied term.

Metadata annotations are terms applied to model elements. Behaviors or constraints described by a metadata annotation must be consistent with the annotated model element. Such annotations define additional behaviors or constraints on the model element, such as a service, entity type, property, function, action, or parameter. For example, a metadata annotation may define ranges of valid values for a particular property. Metadata annotations are applied in CSDL documents describing or referencing an entity model.

Instance annotations are terms applied to a particular instance within an OData payload, such as described in **[OData-JSON]**. An instance annotation can be used to define additional information associated with a particular result, entity, property, or error. For example, whether a property is read-only for a particular instance. Where the same annotation is defined at both the metadata and instance level, the instance-level annotation overrides the annotation specified at the metadata level. Annotations that apply across instances should be specified as metadata annotations.

A *vocabulary* is a schema containing a set of terms where each term is a named metadata extension. Anyone can define a vocabulary (a set of terms) that is scenario-specific or company-specific; more commonly used terms can be published as shared vocabularies such as the OData Core vocabulary **[OData-VocCore]**.

A term can be used to:

- Extend model elements and type instances with additional information.
- Map instances of annotated structured types to an interface defined by the term type; i.e. annotations allow viewing instances of a structured type as instances of a differently structured type specified by the applied term.

A service SHOULD NOT require a client to interpret annotations. Clients SHOULD ignore invalid or unknown terms and silently treat unexpected or invalid values (including invalid type, invalid literal expression, invalid targets, etc.) as an unknown value for the term. Unknown or invalid annotations should never result in an error, as long as the payload remains well-formed.

Example 38: the `Product` entity type is extended with a `DisplayName` by a metadata annotation that binds the term `DisplayName` to the value of the property `Name`. The `Product` entity type also includes an annotation that allows its instances to be viewed as instances of the type specified by the term `SearchResult`

```
"Product": {
  "$Kind": "EntityType",
  "$Key": [
    "ID"
  ],
  "ID": {
    "$Type": "Edm.Int32"
  },
  "Name": {
    "$Nullable": true
  },
  "Description": {
    "$Nullable": true
  },
  "@UI.DisplayName": {
    "$Path": "Name"
  },
  "@SearchVocabulary.SearchResult": {
    "Title": {
      "$Path": "Name"
    },
    "Abstract": {
```

```
      "$Path": "Description"
    },
    "Url": {
      "$Apply": [
        "Products(",
        {
          "$Path": "ID"
        },
        ")"
      ],
      "$Function": "odata.concat"
    }
  }
}
```

14.1 Term

A term allows annotating a model element or OData resource representation with additional data.

The term's name is a simple identifier that MUST be unique within its schema.

The term's type MUST be a type in scope, or a collection of a type in scope.

Term Object

A term is represented as a member of the schema object whose name is the unqualified name of the term and whose value is an object.

The term object MUST contain the member `$Kind` with a string value of `Term`.

It MAY contain the members `$Type`, `$Collection`, `$AppliesTo`, `$Nullable`, `$MaxLength`, `$Precision`, `$Scale`, `$SRID`, and `$DefaultValue`, as well as `$Unicode` for 4.01 and greater payloads.

It MAY contain annotations.

`$Type` and `$Collection`

For single-valued terms the value of `$Type` is the qualified name of the term's type.

For collection-valued terms the value of `$Type` is the qualified name of the term's item type, and the member `$Collection` MUST be present with the literal value `true`.

Absence of the `$Type` member means the type is `Edm.String`.

`$DefaultValue`

The value of `$DefaultValue` is the type-specific JSON representation of the default value of the term, see **[OData-JSON]**.

Note: the `$DefaultValue` member is purely for documentation and isomorphy to **[OData-CSDLXML]**. Annotations in CSDL JSON documents MUST always specify an explicit value.

14.1.1 Specialized Term

A term MAY specialize another term in scope by specifying it as its base term.

When applying a specialized term, the base term MUST also be applied with the same qualifier, and so on until a term without a base term is reached.

`$BaseTerm`

The value of `$BaseTerm` is the qualified name of the base term.

14.1.2 Applicability

The applicability of a term MAY be restricted to a list of model elements. If no list is supplied, the term is not intended to be restricted in its application. The list of model elements MAY be extended in future versions of the vocabulary. As the intended usage may evolve over time, clients SHOULD be prepared for any term to be applied to any model element and SHOULD be prepared to handle unknown values within the list of model constructs. Applicability is expressed using the following symbolic values:

Symbolic Value	Model Element
`Action`	Action
`ActionImport`	Action Import
`Annotation`	Annotation
`Apply`	Application of a client-side function in an annotation
`Cast`	Type Cast annotation expression
`Collection`	Entity Set or collection-valued Property or Navigation Property
`ComplexType`	Complex Type
`EntityContainer`	Entity Container
`EntitySet`	Entity Set
`EntityType`	Entity Type
`EnumType`	Enumeration Type
`Function`	Function
`FunctionImport`	Function Import
`If`	Conditional annotation expression
`Include`	Reference to an Included Schema
`IsOf`	Type Check annotation expression
`LabeledElement`	Labeled Element expression
`Member`	Enumeration Member
`NavigationProperty`	Navigation Property
`Null`	Null annotation expression
`OnDelete`	On-Delete Action of a navigation property
`Parameter`	Action of Function Parameter
`Property`	Property of a structured type
`PropertyValue`	Property value of a Record annotation expression
`Record`	Record annotation expression
`Reference`	Reference to another CSDL document
`ReferentialConstraint`	Referential Constraint of a navigation property

Symbolic Value	Model Element
`ReturnType`	Return Type of an Action or Function
`Schema`	Schema
`Singleton`	Singleton
`Term`	Term
`TypeDefinition`	Type Definition
`UrlRef`	UrlRef annotation expression

$AppliesTo

The value of `$AppliesTo` is an array whose items are strings containing symbolic values from the table above that identify model elements the term is intended to be applied to.

*Example 39: the `IsURL` term can be applied to properties and terms that are of type `Edm.String` (the `Core.Tag` type and the two `Core` terms are defined in **[OData-VocCore]**)*

```
"IsURL": {
  "$Kind": "Term",
  "$Type": "Core.Tag",
  "$DefaultValue": true,
  "$AppliesTo": [
    "Property"
  ],
  "@Core.Description": "Properties and terms annotated with this term MUST
contain a valid URL",
  "@Core.RequiresType": "Edm.String"
}
```

14.2 Annotation

An annotation applies a term to a model element and defines how to calculate a value for the term application. Both term and model element MUST be in scope. Section 14.1.2 specifies which model elements MAY be annotated with a term.

The value of an annotation is specified as an *annotation expression*, which is either a constant expression representing a constant value, or a dynamic expression. The most common construct for assigning an annotation value is a path expression that refers to a property of the same or a related structured type.

Annotation Member

An annotation is represented as a member whose name consists of an at (`@`) character, followed by the qualified name of a term, optionally followed by a hash (`#`) and a qualifier.

The value of the annotation MUST be a constant expression or dynamic expression.

The annotation can be a member of the object representing the model element it annotates, or a second-level member of the `$Annotations` member of a schema object.

An annotation can itself be annotated. Annotations on annotations are represented as a member whose name consists of the annotation name (including the optional qualifier), followed by an at (`@`) character, followed by the qualified name of a term, optionally followed by a hash (`#`) and a qualifier.

Example 40: term `Measures.ISOCurrency`, once applied with a constant value, once with a path value

```
"AmountInReportingCurrency": {
  "$Nullable": true,
  "$Type": "Edm.Decimal",
```

```
  "$Scale": 0,
  "@Measures.ISOCurrency": "USD",
  "@Measures.ISOCurrency@Core.Description": "The parent company's currency"
},
"AmountInTransactionCurrency": {
  "$Nullable": true,
  "$Type": "Edm.Decimal",
  "$Scale": 0,
  "@Measures.ISOCurrency": {
    "$Path": "Currency"
  }
},
"Currency": {
  "$Nullable": true,
  "$MaxLength": 3
}
```

If an entity type or complex type is annotated with a term that itself has a structured type, an instance of the annotated type may be viewed as an "instance" of the term, and the qualified term name may be used as a term-cast segment in path expressions.

Structured types "inherit" annotations from their direct or indirect base types. If both the type and one of its base types is annotated with the same term and qualifier, the annotation on the type completely replaces the annotation on the base type; structured or collection-valued annotation values are not merged. Similarly, properties of a structured type inherit annotations from identically named properties of a base type.

It is up to the definition of a term to specify whether and how annotations with this term propagate to places where the annotated model element is used, and whether they can be overridden. E.g. a "Label" annotation for a UI can propagate from a type definition to all properties using that type definition and may be overridden at each property with a more specific label, whereas an annotation marking a type definition as containing a phone number will propagate to all using properties but may not be overridden.

14.2.1 Qualifier

A term can be applied multiple times to the same model element by providing a qualifier to distinguish the annotations. The qualifier is a simple identifier.

The combination of target model element, term, and qualifier uniquely identifies an annotation.

Example 41: annotation should only be applied to tablet devices

```
"@UI.DisplayName#Tablet": {
  "$Path": "FirstName"
}
```

14.2.2 Target

The target of an annotation is the model element the term is applied to.

The target of an annotation MAY be specified indirectly by "nesting" the annotation within the model element. Whether and how this is possible is described per model element in this specification.

The target of an annotation MAY also be specified directly; this allows defining an annotation in a different schema than the targeted model element.

This external targeting is only possible for model elements that are uniquely identified within their parent, and all their ancestor elements are uniquely identified within their parent:

- Action (single or all overloads)
- Action Import
- Complex Type
- Entity Container

- Entity Set
- Entity Type
- Enumeration Type
- Enumeration Type Member
- Function (single or all overloads)
- Function Import
- Navigation Property (via type, entity set, or singleton)
- Parameter of an action or function (single overloads or all overloads defining the parameter)
- Property (via type, entity set, or singleton)
- Return Type of an action or function (single or all overloads)
- Singleton
- Type Definition

These are the direct children of a schema with a unique name (i.e. except actions and functions whose overloads to not possess a natural identifier), and all direct children of an entity container.

External targeting is possible for actions, functions, their parameters, and their return type, either in a way that applies to all overloads of the action or function or all parameters of that name across all overloads, or in a way that identifies a single overload.

External targeting is also possible for properties and navigation properties of singletons or entities in a particular entity set. These annotations override annotations on the properties or navigation properties targeted via the declaring structured type.

The allowed path expressions are:

- qualified name of schema child
- qualified name of schema child followed by a forward slash and name of child element
- qualified name of structured type followed by zero or more property, navigation property, or type-cast segments, each segment starting with a forward slash
- qualified name of an entity container followed by a segment containing a singleton or entity set name and zero or more property, navigation property, or type-cast segments
- qualified name of an action followed by parentheses containing the qualified name of the binding parameter *type* of a bound action overload to identify that bound overload, or by empty parentheses to identify the unbound overload
- qualified name of a function followed by parentheses containing the comma-separated list of qualified names of the parameter *types* of a bound or unbound function overload in the order of their definition in the function overload
- qualified name of an action or function, optionally followed by parentheses as described in the two previous bullet points to identify a single overload, followed by a forward slash and either a parameter name or `$ReturnType`
- qualified name of an entity container followed by a segment containing an action or function import name, optionally followed by a forward slash and either a parameter name or `$ReturnType`
- One of the preceding, followed by a forward slash, an at (`@`), the qualified name of a term, and optionally a hash (`#`) and the qualifier of an annotation

All qualified names used in a target path MUST be in scope.

Example 42: Target expressions

```
MySchema.MyEntityType
MySchema.MyEntityType/MyProperty
MySchema.MyEntityType/MyNavigationProperty
```

```
MySchema.MyComplexType
MySchema.MyComplexType/MyProperty
MySchema.MyComplexType/MyNavigationProperty
MySchema.MyEnumType
MySchema.MyEnumType/MyMember
MySchema.MyTypeDefinition
MySchema.MyTerm
MySchema.MyEntityContainer
MySchema.MyEntityContainer/MyEntitySet
MySchema.MyEntityContainer/MySingleton
MySchema.MyEntityContainer/MyActionImport
MySchema.MyEntityContainer/MyFunctionImport
MySchema.MyAction
MySchema.MyAction(MySchema.MyBindingType)
MySchema.MyAction()
MySchema.MyFunction
MySchema.MyFunction(MySchema.MyBindingParamType,First.NonBinding.ParamType)
MySchema.MyFunction(First.NonBinding.ParamType,Second.NonBinding.ParamType)
MySchema.MyFunction/MyParameter
MySchema.MyEntityContainer/MyEntitySet/MyProperty
MySchema.MyEntityContainer/MyEntitySet/MyNavigationProperty
MySchema.MyEntityContainer/MyEntitySet/MySchema.MyEntityType/MyProperty
MySchema.MyEntityContainer/MyEntitySet/MySchema.MyEntityType/MyNavProperty
MySchema.MyEntityContainer/MyEntitySet/MyComplexProperty/MyProperty
MySchema.MyEntityContainer/MyEntitySet/MyComplexProperty/MyNavigationProperty
MySchema.MyEntityContainer/MySingleton/MyComplexProperty/MyNavigationProperty
```

14.3 Constant Expression

Constant expressions allow assigning a constant value to an applied term.

14.3.1 Binary

Binary expressions are represented as a string containing the base64url-encoded binary value.

Example 43: base64url-encoded binary value (OData)

```
"@UI.Thumbnail": "T0RhdGE"
```

14.3.2 Boolean

Boolean expressions are represented as the literals `true` or `false`.

Example 44:

```
"@UI.ReadOnly": true
```

14.3.3 Date

Date expressions are represented as a string containing the date value. The value MUST conform to type `xs:date`, see **[XML-Schema-2]**, section 3.3.9. The value MUST also conform to rule `dateValue` in **[OData-ABNF]**, i.e. it MUST NOT contain a time-zone offset.

Example 45:

```
"@vCard.birthDay": "2000-01-01"
```

14.3.4 DateTimeOffset

Datetimestamp expressions are represented as a string containing the timestamp value. The value MUST conform to type `xs:dateTimeStamp`, see **[XML-Schema-2]**, section 3.4.28. The value MUST also conform to rule `dateTimeOffsetValue` in **[OData-ABNF]**, i.e. it MUST NOT contain an end-of-day fragment (24:00:00).

Example 46:

```
"@UI.LastUpdated": "2000-01-01T16:00:00.000Z"
```

14.3.5 Decimal

Decimal expressions are represented as either a number or a string. The special values `INF`, `-INF`, or `NaN` are represented as strings. Numeric values are represented as numbers or strings depending on the media type parameter `IEEE754Compatible`.

Example 47: default representation as a number

```
"@UI.Width": 3.14
```

Example 48: "safe" representation as a string

```
"@UI.Width": "3.14"
```

14.3.6 Duration

Duration expressions are represented as a string containing the duration value. The value MUST conform to type `xs:dayTimeDuration`, see **[XML-Schema-2]**, section 3.4.27.

Example 49:

```
"@task.duration": "P7D"
```

14.3.7 Enumeration Member

Enumeration member expressions are represented as a string containing the numeric or symbolic enumeration value.

Example 50: single value `Red` with numeric value and symbolic value

```
"@self.HasPattern": "1"
```

```
"@self.HasPattern": "Red"
```

Example 51: combined value `Red,Striped` with numeric value 1 + 16 and symbolic value

```
"@self.HasPattern": "17"
```

```
"@self.HasPattern": "Red,Striped"
```

14.3.8 Floating-Point Number

Floating-point expressions are represented as a number or as a string containing one of the special values `INF`, `-INF`, or `NaN`.

Example 52:

```
"@UI.FloatWidth": 3.14
```

```
"@UI.FloatWidth": "INF"
```

14.3.9 Guid

Guid expressions are represented as a string containing the uuid value. The value MUST conform to the rule `guidValue` in **[OData-ABNF]**.

Example 53:

```
"@UI.Id": "21EC2020-3AEA-1069-A2DD-08002B30309D"
```

14.3.10 Integer

Integer expressions are represented as either a number or a string, depending on the media type parameter `IEEE754Compatible`.

Example 54: default representation as a number

```
"@An.Int": 42
```

Example 55: "safe" representation as a string

```
"@A.Very.Long.Int": "9007199254740992"
```

14.3.11 String

String expressions are represented as a JSON string.

Example 56:

```
"@UI.DisplayName": "Product Catalog"
```

14.3.12 Time of Day

Time-of-day expressions are represented as a string containing the time-of-day value. The value MUST conform to the rule `timeOfDayValue` in **[OData-ABNF]**.

Example 57:

```
"@UI.EndTime": "21:45:00"
```

14.4 Dynamic Expression

Dynamic expressions allow assigning a calculated value to an applied term.

14.4.1 Path Expressions

Path expressions allow assigning a value to an applied term or term component. There are two kinds of path expressions:

- A *model path* is used within Annotation Path, Model Element Path, Navigation Property Path, and Property Path expressions to traverse the model of a service and resolves to the model element identified by the path. It allows assigning values to terms or term properties of the built-in types `Edm.AnnotationPath`, `Edm.NavigationPropertyPath`, `Edm.PropertyPath`, and their base types `Edm.AnyPropertyPath` and `Edm.ModelElementPath`.

- An *instance path* is used within a Value Path expression to traverse a graph of type instances and resolves to the value identified by the path. It allows assigning values to terms or term properties of built-in types other than the `Edm.*Path` types, or of any model-defined type.

14.4.1.1 Path Syntax

Model paths and instance paths share a common syntax which is derived from the path expression syntax of URLs, see **[OData-URL]**.

A path MUST be composed of zero or more path segments joined together by forward slashes (`/`).

Paths starting with a forward slash (`/`) are absolute paths, and the first path segment MUST be the qualified name of a model element, e.g. an entity container. The remaining path after the second forward slash is interpreted relative to that model element.

Example 58: absolute path to an entity set

```
/My.Schema.MyEntityContainer/MyEntitySet
```

Paths not starting with a forward slash are interpreted relative to the annotation target, following the rules specified in section "Path Evaluation".

Example 59: relative path to a property

```
Address/City
```

If a path segment is a qualified name, it represents a *type cast*, and the segment MUST be the name of a type in scope. If the type or instance identified by the preceding path part cannot be cast to the specified type, the path expression evaluates to the null value.

Example 60: type-cast segment

```
…/self.Manager/…
```

If a path segment starts with an at (`@`) character, it represents a *term cast*. The at (`@`) character MUST be followed by a qualified name that MAY be followed by a hash (`#`) character and a simple identifier. The qualified name preceding the hash character MUST resolve to a term that is in scope, the simple identifier following the hash sign is interpreted as a qualifier for the term. If the model element or instance identified by the preceding path part has not been annotated with that term (and if present, with that qualifier), the term cast evaluates to the null value. Four special terms are implicitly "annotated" for media entities and stream properties:

- `odata.mediaEditLink`
- `odata.mediaReadLink`
- `odata.mediaContentType`
- `odata.mediaEtag`

Example 61: term-cast segments

```
…/@Capabilities.SortRestrictions/…
```

If a path segment is a simple identifier, it MUST be the name of a child model element of the model element identified by the preceding path part, or a structural or navigation property of the instance identified by the preceding path part. A sequence of navigation segments can traverse multiple CSDL documents. The document containing the path expression only needs to reference the next traversed document to bring the navigation target type into scope, and each traversed document in turn needs to reference only its next document.

A model path MAY contain any number of segments representing collection-valued structural or navigation properties. The result of the expression is the model element reached via this path.

Example 62: property segments in model path

```
…/Orders/Items/Product/…
```

An instance path MUST NOT contain more than one segment representing a collection-valued construct, e.g. an entity set or a collection-valued navigation property that is not followed by a key predicate, or a collection-valued structural property that is not followed by an index segment. The result of the expression is the collection of instances resulting from applying any remaining path segments that operate on a single-valued expression to each instance in the collection-valued segment.

An instance path MAY terminate in a `$count` segment if the previous segment is collection-valued, in which case the path evaluates to the number of items in the collection identified by the preceding segment.

Example 63: property segments in instance path

```
…/Addresses/Street
…/Addresses/$count
```

A model path MAY contain path segments starting with a navigation property, then followed by an at (`@`) character, then followed by the qualified name of a term in scope, and optionally followed by a hash (`#`) character and a simple identifier which is interpreted as a qualifier for the term. If the navigation property has not been annotated with that term (and if present, with that qualifier), the path segment evaluates to the null value. This allows addressing annotations on the navigation property itself; annotations on the entity type specified by the navigation property are addressed via a term-cast segment.

Example 64: model path addressing an annotation on a navigation property

```
…/Items@Capabilities.InsertRestrictions/Insertable
```

An instance path MAY contain path segments starting with an entity set or a collection-valued navigation property, then followed by a key predicate using parentheses-style convention, see **[OData-URL]**. The key values are either primitive literals or instance paths. If the key value is a relative instance path, it is interpreted according to the same rule below as the instance path it is part of, *not* relative to the instance identified by the preceding path part.

Example 65: instance path with entity set and key predicate

```
/self.container/SettingsCollection('FeatureXxx')/IsAvailable
/self.container/Products(ID=ProductID)/Name
```

An instance path MAY contain an index segment immediately following a path segment representing an ordered collection-valued structural property. The index is zero-based and MUST be an integer literal. Negative integers count from the end of the collection, with -1 representing the last item in the collection. Remaining path segments are evaluated relative to the identified item of the collection.

Example 66: instance path with collection-valued structural property and index segment

```
Addresses/1
Addresses/-1/Street
```

14.4.1.2 Path Evaluation

Annotations MAY be embedded within their target, or specified separately, e.g. as part of a different schema, and specify a path to their target model element. The latter situation is referred to as *targeting* in the remainder of this section.

For annotations embedded within or targeting an entity container, the path is evaluated starting at the entity container, i.e. an empty path resolves to the entity container, and non-empty paths MUST start with a segment identifying a container child (entity set, function import, action import, or singleton). The subsequent segments follow the rules for paths targeting the corresponding child element.

For annotations embedded within or targeting an entity set or a singleton, the path is evaluated starting at the entity set or singleton, i.e. an empty path resolves to the entity set or singleton, and non-empty paths MUST follow the rules for annotations targeting the declared entity type of the entity set or singleton.

For annotations embedded within or targeting an entity type or complex type, the path is evaluated starting at the type, i.e. an empty path resolves to the type, and the first segment of a non-empty path MUST be a structural or navigation property of the type, a type cast, or a term cast.

For annotations embedded within a structural or navigation property of an entity type or complex type, the path is evaluated starting at the directly enclosing type. This allows e.g. specifying the value of an annotation on one property to be calculated from values of other properties of the same type. An empty path resolves to the enclosing type, and non-empty paths MUST follow the rules for annotations targeting the directly enclosing type.

For annotations targeting a structural or navigation property of an entity type or complex type, the path is evaluated starting at the *outermost* entity type or complex type named in the target of the annotation, i.e. an empty path resolves to the outermost type, and the first segment of a non-empty path MUST be a structural or navigation property of the outermost type, a type cast, or a term cast.

For annotations embedded within or targeting an action, action import, function, function import, parameter, or return type, the first segment of the path MUST be a parameter name or `$ReturnType`.

14.4.1.3 Annotation Path

The annotation path expression provides a value for terms or term properties that specify the built-in types `Edm.AnnotationPath` or `Edm.ModelElementPath`. Its argument is a model path with the following restriction:

- A non-null path MUST resolve to an annotation.

A term or term property of type `Edm.AnnotationPath` can be annotated with term `Validation.AllowedTerms` (see **[OData-VocValidation]**) if its intended value is an annotation path that ends in a term cast with one of the listed terms.

The value of the annotation path expression is the path itself, not the value of the annotation identified by the path. This is useful for terms that reuse or refer to other terms.

> Annotation path expressions are represented as a string containing a path.

Example 67:

```
"@UI.ReferenceFacet": "Product/Supplier/@UI.LineItem",
"@UI.CollectionFacet#Contacts": [
  "Supplier/@Communication.Contact",
  "Customer/@Communication.Contact"
]
```

14.4.1.4 Model Element Path

The model element path expression provides a value for terms or term properties that specify the built-in type `Edm.ModelElementPath`. Its argument is a model path.

The value of the model element path expression is the path itself, not the instance(s) identified by the path.

> Model element path expressions are represented as a string containing a path.

Example 68:

```
"@org.example.MyFavoriteModelElement": "/self.someAction"
```

14.4.1.5 Navigation Property Path

The navigation property path expression provides a value for terms or term properties that specify the built-in types `Edm.NavigationPropertyPath`, `Edm.AnyPropertyPath`, or `Edm.ModelElementPath`. Its argument is a model path with the following restriction:

- A non-null path MUST resolve to a model element whose type is an entity type, or a collection of entity types, e.g. a navigation property.

The value of the navigation property path expression is the path itself, not the entitiy or collection of entities identified by the path.

> Navigation property path expressions are represented as a string containing a path.

Example 69:

```
"@UI.HyperLink": "Supplier",

"@Capabilities.UpdateRestrictions": {
  "NonUpdatableNavigationProperties": [
    "Supplier",
    "Category"
  ]
}
```

14.4.1.6 Property Path

The property path expression provides a value for terms or term properties that specify one of the built-in types `Edm.PropertyPath`, `Edm.AnyPropertyPath`, or `Edm.ModelElementPath`. Its argument is a model path with the following restriction:

- A non-null path MUST resolve to a model element whose type is a primitive or complex type, an enumeration type, a type definition, or a collection of one of these types.

The value of the property path expression is the path itself, not the value of the structural property or the value of the term cast identified by the path.

> Property path expressions are represented as a string containing a path.

Example 70:

```
"@UI.RefreshOnChangeOf": "ChangedAt",

"@Capabilities.UpdateRestrictions": {
  "NonUpdatableProperties": [
    "CreatedAt",
    "ChangedAt"
  ]
}
```

14.4.1.7 Value Path

The value path expression allows assigning a value by traversing an object graph. It can be used in annotations that target entity containers, entity sets, entity types, complex types, navigation properties of structured types, and structural properties of structured types. Its argument is an instance path.

The value of the path expression is the instance or collection of instances identified by the path.

> **`$Path`**
>
> Path expressions are represented as an object with a single member `$Path` whose value is a string containing a path.

Example 71:

```
"@UI.DisplayName": {
  "$Path": "FirstName"
},

"@UI.DisplayName#second": {
  "$Path": "@vCard.Address#work/FullName"
}
```

14.4.2 Comparison and Logical Operators

Annotations MAY use the following logical and comparison expressions which evaluate to a Boolean value. These expressions MAY be combined and they MAY be used anywhere instead of a Boolean expression.

Operator	Description
Logical Operators	
`And`	Logical and
`Or`	Logical or
`Not`	Logical negation
Comparison Operators	
`Eq`	Equal
`Ne`	Not equal
`Gt`	Greater than
`Ge`	Greater than or equal
`Lt`	Less than
`Le`	Less than or equal
`Has`	Has enumeration flag(s) set
`In`	Is in collection

The `And` and `Or` operators require two operand expressions that evaluate to Boolean values. The `Not` operator requires a single operand expression that evaluates to a Boolean value. For details on null handling for comparison operators see **[OData-URL]**.

The other comparison operators require two operand expressions that evaluate to comparable values.

> #### `$And` and `$Or`
>
> The `And` and `Or` logical expressions are represented as an object with a single member whose value is an array with two annotation expressions. The member name is one of `$And`, or `$Or`.
>
> It MAY contain annotations.
>
> #### `$Not`
>
> Negation expressions are represented as an object with a single member `$Not` whose value is an annotation expression.
>
> It MAY contain annotations.
>
> #### `$Eq`, `$Ne`, `$Gt`, `$Ge`, `$Lt`, `$Le`, `$Has`, and `$In`
>
> All comparison expressions are represented as an object with a single member whose value is an array with two annotation expressions. The member name is one of `$Eq`, `$Ne`, `$Gt`, `$Ge`, `$Lt`, `$Le`, `$Has`, or `$In`.
>
> They MAY contain annotations.

Example 72:

```
{
  "$And": [
```

```
    {
      "$Path": "IsMale"
    },
    {
      "$Path": "IsMarried"
    }
  ]
},
{
  "$Or": [
    {
      "$Path": "IsMale"
    },
    {
      "$Path": "IsMarried"
    }
  ]
},
{
  "$Not": {
    "$Path": "IsMale"
  }
},
{
  "$Eq": [
    null,
    {
      "$Path": "IsMale"
    }
  ]
},
{
  "$Ne": [
    null,
    {
      "$Path": "IsMale"
    }
  ]
},
{
  "$Gt": [
    {
      "$Path": "Price"
    },
    20
  ]
},
{
  "$Ge": [
    {
      "$Path": "Price"
    },
    10
  ]
},
{
  "$Lt": [
    {
      "$Path": "Price"
    },
    20
  ]
},
{
```

```
  "$Le": [
    {
      "$Path": "Price"
    },
    100
  ]
},
{
  "$Has": [
    {
      "$Path": "Fabric"
    },
    "Red"
  ]
},
{
  "$In": [
    {
      "$Path": "Size"
    },
    [
      "XS",
      "S"
    ]
  ]
}
```

14.4.3 Arithmetic Operators

Annotations MAY use the following arithmetic expressions which evaluate to a numeric value. These expressions MAY be combined, and they MAY be used anywhere instead of a numeric expression of the appropriate type. The semantics and evaluation rules for each arithmetic expression is identical to the corresponding arithmetic operator defined in **[OData-URL]**.

Operator	Description
`Add`	Addition
`Sub`	Subtraction
`Neg`	Negation
`Mul`	Multiplication
`Div`	Division (with integer result for integer operands)
`DivBy`	Division (with fractional result also for integer operands)
`Mod`	Modulo

The `Neg` operator requires a single operand expression that evaluates to a numeric value. The other arithmetic operators require two operand expressions that evaluate to numeric values.

> **`$Neg`**
>
> Negation expressions are represented as an object with a single member `$Neg` whose value is an annotation expression.
>
> It MAY contain annotations.

$Add, $Sub, $Mul, $Div, $DivBy, and $Mod

These arithmetic expressions are represented as an object with as single member whose value is an array with two annotation expressions. The member name is one of `$Add`, `$Sub`, `$Neg`, `$Mul`, `$Div`, `$DivBy`, or `$Mod`.

They MAY contain annotations.

Example 73:

```
{
  "$Add": [
    {
      "$Path": "StartDate"
    },
    {
      "$Path": "Duration"
    }
  ]
},
{
  "$Sub": [
    {
      "$Path": "Revenue"
    },
    {
      "$Path": "Cost"
    }
  ]
},
{
  "$Neg": {
    "$Path": "Height"
  }
},
{
  "$Mul": [
    {
      "$Path": "NetPrice"
    },
    {
      "$Path": "TaxRate"
    }
  ]
},
{
  "$Div": [
    {
      "$Path": "Quantity"
    },
    {
      "$Path": "QuantityPerParcel"
    }
  ]
},
{
  "$DivBy": [
    {
      "$Path": "Quantity"
    },
    {
      "$Path": "QuantityPerParcel"
    }
  ]
},
```

```
{
  "$Mod": [
    {
      "$Path": "Quantity"
    },
    {
      "$Path": "QuantityPerParcel"
    }
  ]
}
```

14.4.4 Apply Client-Side Functions

The apply expression enables a value to be obtained by applying a client-side function. The apply expression MAY have operand expressions. The operand expressions are used as parameters to the client-side function.

> **`$Apply`**
>
> Apply expressions are represented as an object with a member `$Apply` whose value is an array of annotation expressions, and a member `$Function` whose value is a string containing the qualified name of the client-side function to be applied.
>
> It MAY contain annotations.

OData defines the following functions. Services MAY support additional functions that MUST be qualified with a namespace other than `odata`. Function names qualified with `odata` are reserved for this specification and its future versions.

14.4.4.1 Canonical Functions

All canonical functions defined in **[OData-URL]** can be used as client-side functions, qualified with the namespace `odata`. The semantics of these client-side functions is identical to their counterpart function defined in **[OData-URL]**.

For example, the `odata.concat` client-side function takes two or more expressions as arguments. Each argument MUST evaluate to a primitive or enumeration type. It returns a value of type `Edm.String` that is the concatenation of the literal representations of the results of the argument expressions. Values of primitive types other than `Edm.String` are represented according to the appropriate alternative in the `primitiveValue` rule of **[OData-ABNF]**, i.e. `Edm.Binary` as `binaryValue`, `Edm.Boolean` as `booleanValue` etc.

Example 74:

```
"@UI.DisplayName": {
  "$Apply": [
    "Product: ",
    {
      "$Path": "ProductName"
    },
    " (",
    {
      "$Path": "Available/Quantity"
    },
    " ",
    {
      "$Path": "Available/Unit"
    },
    " available)"
  ],
  "$Function": "odata.concat"
}
```

`ProductName` is of type `String`, `Quantity` in complex type `Available` is of type `Decimal`, and `Unit` in `Available` is of type enumeration, so the result of the `Path` expression is represented as the member name of the enumeration value.

14.4.4.2 Function `odata.fillUriTemplate`

The `odata.fillUriTemplate` client-side function takes two or more expressions as arguments and returns a value of type `Edm.String`.

The first argument MUST be of type `Edm.String` and specifies a URI template according to **[RFC6570]**, the other arguments MUST be labeled element expressions. Each labeled element expression specifies the template parameter name as its name and evaluates to the template parameter value.

[RFC6570] defines three kinds of template parameters: simple values, lists of values, and key-value maps.

Simple values are represented as labeled element expressions that evaluate to a single primitive value. The literal representation of this value according to **[OData-ABNF]** is used to fill the corresponding template parameter.

Lists of values are represented as labeled element expressions that evaluate to a collection of primitive values.

Key-value maps are represented as labeled element expressions that evaluate to a collection of complex types with two properties that are used in lexicographic order. The first property is used as key, the second property as value.

Example 75: assuming there are no special characters in values of the `Name` property of the `Actor` entity

```
{
  "$Apply": [
    "http://host/someAPI/Actors/{actorName}/CV",
    {
      "$LabeledElement": {
        "$Path": "Actor/Name"
      },
      "$Name": "self.actorName"
    }
  ],
  "$Function": "odata.fillUriTemplate"
}
```

14.4.4.3 Function `odata.matchesPattern`

The `odata.matchesPattern` client-side function takes two string expressions as arguments and returns a Boolean value.

The function returns true if the second expression evaluates to an **[ECMAScript]** (JavaScript) regular expression and the result of the first argument expression matches that regular expression, using syntax and semantics of **[ECMAScript]** regular expressions.

Example 76: all non-empty `FirstName` values not containing the letters `b`, `c`, or `d` evaluate to `true`

```
{
  "$Apply": [
    {
      "$Path": "FirstName"
    },
    "^[^b-d]+$"
  ],
  "$Function": "odata.matchesPattern"
}
```

14.4.4.4 Function `odata.uriEncode`

The `odata.uriEncode` client-side function takes one argument of primitive type and returns the URL-encoded OData literal that can be used as a key value in OData URLs or in the query part of OData URLs.

Note: string literals are surrounded by single quotes as required by the paren-style key syntax.

Example 77:

```
{
  "$Apply": [
    "http://host/service/Genres({genreName})",
    {
      "$LabeledElement": {
        "$Apply": [
          {
            "$Path": "NameOfMovieGenre"
          }
        ],
        "$Function": "odata.uriEncode"
      },
      "$Name": "self.genreName"
    }
  ],
  "$Function": "odata.fillUriTemplate"
}
```

14.4.5 Cast

The cast expression casts the value obtained from its single child expression to the specified type. The cast expression follows the same rules as the `cast` canonical function defined in **[OData-URL]**.

> **`$Cast`**
>
> Cast expressions are represented as an object with a member `$Cast` whose value is an annotation expression, a member `$Type` whose value is a string containing the qualified type name, and optionally a member `$Collection` with a value of `true`.
>
> It MAY contain annotations.
>
> If the specified type is a primitive type or a collection of primitive types, the facet members `$MaxLength`, `$Unicode`, `$Precision`, `$Scale`, and `$SRID` MAY be specified if applicable to the specified primitive type. If the facet members are not specified, their values are considered unspecified.

Example 78:

```
"@UI.Threshold": {
  "$Cast": {
    "$Path": "Average"
  },
  "$Type": "Edm.Decimal"
}
```

14.4.6 Collection

The collection expression enables a value to be obtained from zero or more item expressions. The value calculated by the collection expression is the collection of the values calculated by each of the item expressions. The values of the child expressions MUST all be type compatible.

> Collection expressions are represented as arrays with one array item per item expression within the collection expression.

Example 79:

```
"@seo.SeoTerms": [
  "Product",
  "Supplier",
  "Customer"
]
```

14.4.7 If-Then-Else

The if-then-else expression enables a value to be obtained by evaluating a *condition expression*. It MUST contain exactly three child expressions. There is one exception to this rule: if and only if the if-then-else expression is an item of a collection expression, the third child expression MAY be omitted, reducing it to an if-then expression. This can be used to conditionally add an element to a collection.

The first child expression is the condition and MUST evaluate to a Boolean result, e.g. the comparison and logical operators can be used.

The second and third child expressions are evaluated conditionally. The result MUST be type compatible with the type expected by the surrounding expression.

If the first expression evaluates to `true`, the second expression MUST be evaluated and its value MUST be returned as the result of the if-then-else expression. If the first expression evaluates to `false` and a third child element is present, it MUST be evaluated and its value MUST be returned as the result of the if-then-else expression. If no third expression is present, nothing is added to the surrounding collection.

`$If`

Conditional expressions are represented as an object with a member `$If` whose value is an array of two or three annotation expressions.

It MAY contain annotations.

Example 80: the condition is a value path expression referencing the Boolean property `IsFemale`,whose value then determines the value of the `$If` expression

```
"@person.Gender": {
  "$If": [
    {
      "$Path": "IsFemale"
    },
    "Female",
    "Male"
  ]
}
```

14.4.8 Is-Of

The is-of expression checks whether the value obtained from its single child expression is compatible with the specified type. It returns `true` if the child expression returns a type that is compatible with the specified type, and `false` otherwise.

`$IsOf`

Is-of expressions are represented as an object with a member `$IsOf` whose value is an annotation expression, a member `$Type` whose value is a string containing an qualified type name, and optionally a member `$Collection` with a value of `true`.

It MAY contain annotations.

If the specified type is a primitive type or a collection of primitive types, the facet members `$MaxLength`, `$Unicode`, `$Precision`, `$Scale`, and `$SRID` MAY be specified if applicable to the specified primitive type. If the facet members are not specified, their values are considered unspecified.

Example 81:

```
"@Self.IsPreferredCustomer": {
  "$IsOf": {
    "$Path": "Customer"
  },
  "$Type": "self.PreferredCustomer"
}
```

14.4.9 Labeled Element

The labeled element expression assigns a name to its single child expression. The value of the child expression can then be reused elsewhere with a labeled element reference expression.

A labeled element expression MUST contain exactly one child expression. The value of the child expression is also the value of the labeled element expression.

A labeled element expression MUST provide a simple identifier value as its name that MUST be unique within the schema containing the expression.

> **`$LabeledElement`**
>
> Labeled element expressions are represented as an object with a member `$LabeledElement` whose value is an annotation expression, and a member `$Name` whose value is a string containing the labeled element's name.
>
> It MAY contain annotations.

Example 82:

```
"@UI.DisplayName": {
  "$LabeledElement": {
    "$Path": "FirstName"
  },
  "$Name": "CustomerFirstName"
}
```

14.4.10 Labeled Element Reference

The labeled element reference expression MUST specify the qualified name of a labeled element expression in scope and returns the value of the identified labeled element expression as its value.

> **`$LabeledElementReference`**
>
> Labeled element reference expressions are represented as an object with a member `$LabeledElementReference` whose value is a string containing an qualified name.

Example 83:

```
"@UI.DisplayName": {
  "$LabeledElementReference": "self.CustomerFirstName"
}
```

14.4.11 Null

The null expression indicates the absence of a value. The null expression MAY be annotated.

> Null expressions that do not contain annotations are represented as the literal `null`.

Example 84:

```
"@UI.DisplayName": null,
```

`$Null`

Null expression containing annotations are represented as an object with a member `$Null` whose value is the literal `null`.

Example 85:

```
"@UI.Address": {
  "$Null": null,
  "@self.Reason": "Private"
}
```

14.4.12 Record

The record expression enables a new entity type or complex type instance to be constructed.

A record expression MAY specify the structured type of its result, which MUST be an entity type or complex type in scope. If not explicitly specified, the type is derived from the expression's context.

A record expression contains zero or more property value expressions. For each single-valued structural or navigation property of the record expression's type that is neither nullable nor specifies a default value a property value expression MUST be provided. The only exception is if the record expression is the value of an annotation for a term that has a base term whose type is structured and directly or indirectly inherits from the type of its base term. In this case, property values that already have been specified in the annotation for the base term or its base term etc. need not be specified again.

For collection-valued properties the absence of a property value expression is equivalent to specifying an empty collection as its value.

Record expressions are represented as objects with one member per property value expression. The member name is the property name, and the member value is the property value expression.

The type of a record expression is represented as the `@type` control information, see **[OData-JSON]**.

It MAY contain annotations for itself and its members. Annotations for record members are prefixed with the member name.

Example 86: this annotation "morphs" the entity type from example 8 into a structured type with two structural properties `GivenName` and `Surname` and two navigation properties `DirectSupervisor` and `CostCenter`. The first three properties simply rename properties of the annotated entity type, the fourth adds a calculated navigation property that is pointing to a different service

```
"@person.Employee": {
  "@type": "https://example.org/vocabs/person#org.example.person.Manager",
  "@Core.Description": "Annotation on record",
  "GivenName": {
    "$Path": "FirstName"
  },
  "GivenName@Core.Description": "Annotation on record member",
  "Surname": {
    "$Path": "LastName"
  },
  "DirectSupervisor": {
    "$Path": "Manager"
  },
  "CostCenter": {
    "$UrlRef": {
      "$Apply": [
        "http://host/anotherservice/CostCenters('{ccid}')",
        {
          "$LabeledElement": {
            "$Path": "CostCenterID"
          },
          "$Name": "ccid"
```

```
        }
      ],
      "$Function": "odata.fillUriTemplate"
    }
  }
}
```

14.4.13 URL Reference

The URL reference expression enables a value to be obtained by sending a `GET` request.

The URL reference expression MUST contain exactly one expression of type `Edm.String`. Its value is treated as a URL that MAY be relative or absolute; relative URLs are relative to the URL of the document containing the URL reference expression, or relative to a base URL specified in a format-specific way.

The response body of the `GET` request MUST be returned as the result of the URL reference expression. The result of the URL reference expression MUST be type compatible with the type expected by the surrounding expression.

> **`$UrlRef`**
>
> URL reference expressions are represented as an object with a single member `$UrlRef` whose value is an annotation expression.
>
> It MAY contain annotations.

Example 87:

```
"@org.example.person.Supplier": {
  "$UrlRef": {
    "$Apply": [
      "http://host/service/Suppliers({suppID})",
      {
        "$LabeledElement": {
          "$Apply": [
            {
              "$Path": "SupplierId"
            }
          ],
          "$Function": "odata.uriEncode"
        },
        "$Name": "suppID"
      }
    ],
    "$Function": "odata.fillUriTemplate"
  }
},

"@Core.LongDescription#element": {
  "$UrlRef": "http://host/wiki/HowToUse"
}
```

15 Identifier and Path Values

15.1 Namespace

A namespace is a dot-separated sequence of simple identifiers with a maximum length of 511 Unicode characters (code points).

15.2 Simple Identifier

A simple identifier is a Unicode character sequence with the following restrictions:

- It consists of at least one and at most 128 Unicode characters (code points).
- The first character MUST be the underscore character (U+005F) or any character in the Unicode category “Letter (L)” or “Letter number (Nl)”.
- The remaining characters MUST be the underscore character (U+005F) or any character in the Unicode category “Letter (L)”, “Letter number (Nl)”, “Decimal number (Nd)”, “Non-spacing mark (Mn)”, “Combining spacing mark (Mc)”, “Connector punctuation (Pc)”, and “Other, format (Cf)”.

Non-normatively speaking it starts with a letter or underscore, followed by at most 127 letters, underscores or digits.

15.3 Qualified Name

For model elements that are direct children of a schema: the namespace or alias of the schema that defines the model element, followed by a dot and the name of the model element, see rule `qualifiedTypeName` in **[OData-ABNF]**.

For built-in primitive types: the name of the type, prefixed with `Edm` followed by a dot.

15.4 Target Path

Target paths are used to refer to other model elements.

The allowed path expressions are:

- The qualified name of an entity container, followed by a forward slash and the name of a container child element
- The target path of a container child followed by a forward slash and one or more forward-slash separated property, navigation property, or type-cast segments

Example 88: Target expressions

```
MySchema.MyEntityContainer/MyEntitySet
MySchema.MyEntityContainer/MySingleton
MySchema.MyEntityContainer/MySingleton/MyContainmentNavigationProperty
MySchema.MyEntityContainer/MySingleton/My.EntityType/MyContainmentNavProperty
MySchema.MyEntityContainer/MySingleton/MyComplexProperty/MyContainmentNavProp
```

16 CSDL Examples

Following are two basic examples of valid EDM models as represented in CSDL JSON. These examples demonstrate many of the topics covered above.

16.1 Products and Categories Example

Example 89:

```
{
  "$Version": "4.0",
  "$EntityContainer": "ODataDemo.DemoService",
  "$Reference": {
    "https://oasis-tcs.github.io/odata-
vocabularies/vocabularies/Org.OData.Core.V1.json": {
      "$Include": [
        {
          "$Namespace": "Org.OData.Core.V1",
          "$Alias": "Core",
          "@Core.DefaultNamespace": true
        }
      ]
    },
    "https://oasis-tcs.github.io/odata-
vocabularies/vocabularies/Org.OData.Measures.V1.json": {
      "$Include": [
        {
          "$Namespace": "Org.OData.Measures.V1",
          "$Alias": "Measures"
        }
      ]
    }
  },
  "ODataDemo": {
    "$Alias": "self",
    "@Core.DefaultNamespace": true,
    "Product": {
      "$Kind": "EntityType",
      "$HasStream": true,
      "$Key": [
        "ID"
      ],
      "ID": {},
      "Description": {
        "$Nullable": true,
        "@Core.IsLanguageDependent": true
      },
      "ReleaseDate": {
        "$Nullable": true,
        "$Type": "Edm.Date"
      },
      "DiscontinuedDate": {
        "$Nullable": true,
        "$Type": "Edm.Date"
      },
      "Rating": {
        "$Nullable": true,
        "$Type": "Edm.Int32"
      },
      "Price": {
        "$Nullable": true,
        "$Type": "Edm.Decimal",
```

```
        "@Measures.ISOCurrency": {
          "$Path": "Currency"
        }
      },
      "Currency": {
        "$Nullable": true,
        "$MaxLength": 3
      },
      "Category": {
        "$Kind": "NavigationProperty",
        "$Type": "self.Category",
        "$Partner": "Products"
      },
      "Supplier": {
        "$Kind": "NavigationProperty",
        "$Nullable": true,
        "$Type": "self.Supplier",
        "$Partner": "Products"
      }
    },
    "Category": {
      "$Kind": "EntityType",
      "$Key": [
        "ID"
      ],
      "ID": {
        "$Type": "Edm.Int32"
      },
      "Name": {
        "@Core.IsLanguageDependent": true
      },
      "Products": {
        "$Kind": "NavigationProperty",
        "$Partner": "Category",
        "$Collection": true,
        "$Type": "self.Product",
        "$OnDelete": "Cascade"
      }
    },
    "Supplier": {
      "$Kind": "EntityType",
      "$Key": [
        "ID"
      ],
      "ID": {},
      "Name": {
        "$Nullable": true
      },
      "Address": {
        "$Type": "self.Address"
      },
      "Concurrency": {
        "$Type": "Edm.Int32"
      },
      "Products": {
        "$Kind": "NavigationProperty",
        "$Partner": "Supplier",
        "$Collection": true,
        "$Type": "self.Product"
      }
    },
    "Country": {
      "$Kind": "EntityType",
      "$Key": [
```



```
        "Code"
      ],
      "Code": {
        "$MaxLength": 2
      },
      "Name": {
        "$Nullable": true
      }
    },
    "Address": {
      "$Kind": "ComplexType",
      "Street": {
        "$Nullable": true
      },
      "City": {
        "$Nullable": true
      },
      "State": {
        "$Nullable": true
      },
      "ZipCode": {
        "$Nullable": true
      },
      "CountryName": {
        "$Nullable": true
      },
      "Country": {
        "$Kind": "NavigationProperty",
        "$Nullable": true
        "$Type": "self.Country",
        "$ReferentialConstraint": {
          "CountryName": "Name"
        }
      }
    },
    "ProductsByRating": [
      {
        "$Kind": "Function",
        "$Parameter": [
          {
            "$Name": "Rating",
            "$Nullable": true,
            "$Type": "Edm.Int32"
          }
        ],
        "$ReturnType": {
          "$Collection": true,
          "$Type": "self.Product"
        }
      }
    ],
    "DemoService": {
      "$Kind": "EntityContainer",
      "Products": {
        "$Collection": true,
        "$Type": "self.Product",
        "$NavigationPropertyBinding": {
          "Category": "Categories"
        }
      },
      "Categories": {
        "$Collection": true,
        "$Type": "self.Category",
        "$NavigationPropertyBinding": {
```



```
          "Products": "Products"
        },
        "@Core.Description": "Product Categories"
      },
      "Suppliers": {
        "$Collection": true,
        "$Type": "self.Supplier",
        "$NavigationPropertyBinding": {
          "Products": "Products",
          "Address/Country": "Countries"
        },
        "@Core.OptimisticConcurrency": [
          "Concurrency"
        ]
      },
      "Countries": {
        "$Collection": true,
        "$Type": "self.Country"
      },
      "MainSupplier": {
        "$Type": "self.Supplier",
        "$NavigationPropertyBinding": {
          "Products": "Products"
        },
        "@Core.Description": "Primary Supplier"
      },
      "ProductsByRating": {
        "$EntitySet": "Products",
        "$Function": "self.ProductsByRating"
      }
    }
  }
}
```

16.2 Annotations for Products and Categories Example

Example 90:

```
{
  "$Version": "4.01",
  "$Reference": {
    "http://host/service/$metadata": {
      "$Include": [
        {
          "$Namespace": "ODataDemo",
          "$Alias": "target"
        }
      ]
    },
    "http://somewhere/Vocabulary/V1": {
      "$Include": [
        {
          "$Namespace": "Some.Vocabulary.V1",
          "$Alias": "Vocabulary1"
        }
      ]
    }
  },
  "External.Annotations": {
    "$Annotations": {
      "target.Supplier": {
        "@Vocabulary1.EMail": null,
        "@Vocabulary1.AccountID": {
          "$Path": "ID"
```

```
        },
        "@Vocabulary1.Title": "Supplier Info",
        "@Vocabulary1.DisplayName": {
          "$Apply": [
            {
              "$Path": "Name"
            },
            " in ",
            {
              "$Path": "Address/CountryName"
            }
          ],
          "$Function": "odata.concat"
        }
      },
      "target.Product": {
        "@Vocabulary1.Tags": [
          "MasterData"
        ]
      }
    }
  }
}
```


17 Conformance

Conforming services MUST follow all rules of this specification document for the types, sets, functions, actions, containers and annotations they expose.

In addition, conforming services MUST NOT return 4.01 CSDL constructs for requests made with `OData-MaxVersion:4.0`.

Specifically, they

1. MUST NOT include properties in derived types that overwrite a property defined in the base type
2. MUST NOT include `Edm.Untyped`
3. MUST NOT use path syntax added with 4.01
4. MUST NOT use `Edm.ModelElementPath` and `Edm.AnyPropertyPath`
5. MUST NOT specify referential constraints to complex types and navigation properties
6. MUST NOT include a non-abstract entity type with no inherited or defined entity key
7. MUST NOT include the `Core.DefaultNamespace` annotation on included schemas
8. MUST NOT return the Unicode facet for terms, parameters, and return types
9. MUST NOT include collections of `Edm.ComplexType` or `Edm.Untyped`
10. MUST NOT specify a key as a property of a related entity
11. SHOULD NOT include new/unknown values for `$AppliesTo`
12. MAY include new CSDL annotations

In addition, OData 4.01 services:

13. SHOULD NOT have identifiers within a uniqueness scope (e.g. a schema, a structural type, or an entity container) that differ only by case

Conforming clients MUST be prepared to consume a model that uses any or all constructs defined in this specification, including custom annotations, and MUST ignore constructs not defined in this version of the specification.

Appendix A. Acknowledgments

The work of the OpenUI5 team on the OData V4 Metadata JSON Format, see **[OpenUI5]**, is gratefully acknowledged, especially the contributions of

- Thomas Chadzelek (SAP SE)
- Jens Ittel (SAP SE)
- Patric Ksinsik (SAP SE)

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

Appendix B. Table of JSON Objects and Members

Appendix C. Revision History

Revision	Date	Editor	Changes Made
Working Draft 01	2016-11-16	Ralf Handl	Initial version
Committee Specification Draft 01	2017-06-08	Michael Pizzo Ralf Handl	Integrated 4.01 features
Committee Specification Draft 02	2017-09-22	Michael Pizzo Ralf Handl	Incorporated review feedback Changed defaults of `$Nullable`, `$Scale`, and `$Precision`
Committee Specification Draft 03	2017-11-10	Michael Pizzo Ralf Handl	Incorporated review feedback Stable order of action and function parameters
Committee Specification 01	2017-12-19	Michael Pizzo Ralf Handl	Non-Material Changes
Committee Specification Draft 04	2019-06-21	Michael Pizzo Ralf Handl	External targeting for annotations on action/function overloads, parameters, and return types Key and index segments for path expressions in annotations Nullable singletons Simplified syntax of entity container children and constant annotation expressions
Committee Specification Draft 05	2019-09-26	Michael Pizzo Ralf Handl	Redefining entity sets and singletons when extending entity containers
Committee Specification 02	2019-11-05	Michael Pizzo Ralf Handl	Non-material changes
Candidate OASIS Standard 01	2020-01-15	Michael Pizzo Ralf Handl	Non-material changes