OData Version 4.01. Part 3: Common Schema Definition Language (CSDL)

Committee Specification Draft 01 / Public Review Draft 01

08 December 2016

Specification URIs

This version:

http://docs.oasis-open.org/odata/odata/v4.01/csprd01/part3-csdl/odata-v4.01-csprd01-part3-csdl.docx (Authoritative)
http://docs.oasis-open.org/odata/odata/v4.01/csprd01/part3-csdl/odata-v4.01-csprd01-part3-csdl.html
http://docs.oasis-open.org/odata/odata/v4.01/csprd01/part3-csdl/odata-v4.01-csprd01-part3-csdl.pdf

Previous version:

N/A

Latest version:

http://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part3-csdl.docx (Authoritative)
http://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part3-csdl.html
http://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part3-csdl.pdf

Technical Committee:

OASIS Open Data Protocol (OData) TC

Chairs:

Ralf Handl (ralf.handl@sap.com), SAP SE
Ram Jeyaraman (Ram.Jeyaraman@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:

- *OData Version 4.01. Part 1: Protocol*. http://docs.oasis-open.org/odata/odata/v4.01/csprd01/part1-protocol/odata-v4.01-csprd01-part1-protocol.html.
- *OData Version 4.01. Part 2: URL Conventions*. http://docs.oasis-open.org/odata/odata/v4.01/csprd01/part2-url-conventions/odata-v4.01-csprd01-part2-url-conventions.html.
- *OData Version 4.01. Part 3: Common Schema Definition Language (CSDL)* (this document). http://docs.oasis-open.org/odata/odata/v4.01/csprd01/part3-csdl/odata-v4.01-csprd01-part3-csdl.html.
- ABNF components: OData ABNF Construction Rules Version 4.0 and OData ABNF Test Cases. http://docs.oasis-open.org/odata/odata/v4.01/csprd01/abnf/.

Related work:

This specification replaces or supersedes:

- *OData Version 4.0 Part 3: Common Schema Definition Language (CSDL)*. Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. 24 February 2014. OASIS Standard. http://docs.oasis-open.org/odata/odata/v4.0/os/part3-csdl/odata-v4.0-os-part3-csdl.html. Latest version: http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part3-csdl.html.

This specification is related to:

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

Abstract:

OData services are described by an Entity Data 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 describes CSDL concepts that are common across the various representations such as using XML or JSON formats.

Status:

This document was last revised or approved by the OASIS Open Data Protocol (OData) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document. 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/.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (https://www.oasis-open.org/committees/odata/ipr.php).

Citation format:

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

[OData-Part3]

OData Version 4.01. Part 3: Common Schema Definition Language (CSDL). Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. 08 December 2016. OASIS Committee Specification Draft 01 / Public Review Draft 01. http://docs.oasis-open.org/odata/odata/v4.01/csprd01/part3-csdl/odata-v4.01-csprd01-part3-csdl.html. Latest version: http://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part3-csdl.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

An OData service is based on an entity model described with the Common Schema Definition Language (CSDL). This document specifies CSDL independent of the format used to represent a CDSL document.

CSDL can be represented with the Extensible Markup Language (XML), see **[OData-CSDLXML]**.

CSDL can alternatively be represented with the JavaScript Object Notation (JSON), see **[OData-CSDLJSON]**.

1.1 Terminology

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

1.2 Normative References

[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-CSDLJSON] OData Common Schema Definition Language (CSDL) JSON Representation Version 4.01. See link in “Related work” 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 “Additional artifacts” section on cover page.

[OData-URL] *OData Version 4.01 Part 2: URL Conventions*.
See link in “Additional artifacts” section on cover page.

[OData-VocCore] *OData Vocabularies Version 4.0: Core Vocabulary.*
See link in “Related work” section on cover page.

[RFC2119] Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, 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.

[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/.

1.3 Typographical Conventions

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

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

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

Example 1: text describing an example uses this paragraph style

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

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

All other text is normative unless otherwise labeled.

2 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 physical locations.

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 referenced documents.

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 namespace-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 elements defined in schemas referenced by their defining schema.

2.1 References

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` header, defined **[OData-Protocol]**, SHOULD be used when retrieving the referenced schema document.

2.2 Included Schemas

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 SimpleIdentifier value MAY be specified as an alias for the schema that can be used in qualified names instead of the namespace. It only provides a more convenient notation. Every model element that can be used via an alias-qualified name can alternatively also be used via its full namespace-qualified name. An alias allows a short string to be substituted for a long namespace. For instance, 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.

Aliases are document-global, so all schemas defined within or included into a document MUST have different aliases.

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 has to define its own aliases when including schemas.

2.3 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.

2.4 Nominal Types

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

2.5 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.

2.6 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

Type	Meaning
Edm.Byte	Unsigned 8-bit integer
Edm.Date	Date without a time-zone offset
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 UTF-8 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` and negative years.

`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 a function or action. `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 5.3.

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.

2.7 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.
  - cannot be used as the underlying type of a type definition or enumeration type.
- `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.
  - cannot be used as the return type of a 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.

Vocabulary terms can, in addition, use

- `Edm.AnnotationPath`

- `Edm.PropertyPath`
- `Edm.NavigationPropertyPath`
- `Edm.AnyPropertyPath` (`Edm.PropertyPath` or `Edm.NavigationPropertyPath`)
- `Edm.AnyPath` (`Edm.AnyPropertyPath` or `Edm.AnnotationPath`)

as the type of a primitive term, or the type of a property of a complex type that is exclusively used as the type of a term.

2.8 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.

3 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 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. See Nominal Types for more detail.

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

3.1 Alias

A schema MAY define an alias which MUST be a SimpleIdentifier. The alias can be used instead of the namespace within qualified names to identify model elements.

Aliases are document-global, so all schemas defined within or included into a document MUST have different aliases. 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`.

4 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.

A key MAY be specified if the entity type does not specify a base type that already has a key declared.

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.

An open entity type allows properties to be dynamically added to instances of the type.

4.1 Name

The entity type's name is a SimpleIdentifier that MUST be unique within its namespace.

4.2 Key

An entity is uniquely identified within an entity set by its key. 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 or a non-nullable primitive property of a single-valued, non-nullable complex or navigation property (recursively) of the entity type. Navigation properties MAY only be used in OData 4.01 responses.

If the key property is a property of a complex or navigation property (recursively), the key MUST specify an alias for that property that MUST be a SimpleIdentifier 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.

4.3 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.

4.4 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.

4.5 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]**.

4.6 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. 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]**.

5 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.

5.1 Name

The property's name MUST be a SimpleIdentifier. 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.

5.2 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-CoreVoc]**, to specify that it supports a stable ordering.

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

5.3 Facets

Facets modify or constrain the acceptable values of a property.

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

5.3.1 Nullable

If true, `null` is an allowed value.

5.3.2 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.

Instead of an integer value the constant `max` MAY be specified as a shorthand for the maximum length supported for the type by the service.

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

5.3.3 Precision

For a decimal value: the maximum number of significant decimal digits of the property's value; it MUST be a positive integer. If no value is specified, the decimal property has arbitrary precision.

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. If no value is specified, the temporal property has a precision of zero.

Note: service designers 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.

5.3.4 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` MAY be used to specify 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` MAY be used to specify 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 has to precede the decimal point.

The value of `Scale` MUST be less than or equal to the value of `Precision`. If no value is specified, the `Scale` facet defaults to zero.

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.

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

Example 3: `Precision=2` equals `Scale`.
Allowed values: 0.23, 0,7, not allowed values: 1.23, 1.2.

Example 4: `Precision=3` and `Scale=variable`.
Allowed values: 0.123, 1.23, 0.23, 0.7, 123 and 12.3, not allowed would be: 12.34, 1234 and 123.4 due to the limited precision.

Example 5: `Precision=7` and `Scale=floating`.
Allowed values: -1.234567e3, 1e-101, 9.999999e96, not allowed would be: 1e-102 and 1e97 due to the limited precision.

5.3.5 Unicode

The value `true` indicates that the property might contain and accept string values with Unicode characters 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`.

5.3.6 SRID

For a geometry or geography 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]**.

5.3.7 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.

Default values of type `Edm.String` MUST be represented according to the escaping rules for character data in the CSDL representation. Values of other primitive types MUST be represented according to the appropriate alternative in the `primitiveValue` rule defined in **[OData-ABNF]**, i.e. `Edm.Binary` as `binaryValue`, `Edm.Boolean` as `booleanValue` etc.

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

6 Navigation Property

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

6.1 Name

The navigation property's name MUST be a SimpleIdentifier. 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.

6.2 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-CoreVoc]**, to specify that it supports a stable ordering.

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

6.3 Nullable

If true or not specified for a single-valued navigation property, instances of 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.

6.4 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.

6.5 Containment Navigation Property

A navigation property MAY indicate that instances 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-CoreVoc]**), 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.

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. 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.

6.6 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 *dependent entity* declaring the navigation property) MUST have the same value as the *principal property* (the referenced property declared on the *principal entity* 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 is an entity type, then the dependent property must reference the same entity.

If the principle property is a complex type, then the dependent property must reference a complex type 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.

6.7 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.

7 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.

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.

An open complex type allows properties to be dynamically added to instances of the type.

7.1 Name

The complex type's name is a SimpleIdentifier that MUST be unique within its namespace.

7.2 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.

7.3 Abstract Complex Type

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

7.4 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]**.

8 Enumeration Type

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

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.

8.1 Name

The enumeration type's name is a SimpleIdentifier that MUST be unique within its namespace.

8.2 Underlying 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.

8.3 Flags Enumeration Types

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

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

8.4 Enumeration Type Members

Enumeration type values consist of discrete members.

Each member is identified by its name, a SimpleIdentifier that MUST be unique within the enumeration type.

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 value compare as equal, and members with the same value can be used interchangeably.

Enumeration members are sorted by their numeric value.

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

9 Type Definition

A type definition defines a specialization of one of the primitive types.

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.

9.1 Name

The type definition's name is a SimpleIdentifier that MUST be unique within its namespace.

9.2 Underlying Type

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

9.2.1 Type Definition Facets

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.

Annotations MAY be applied to a type definition, and are considered applied wherever the type definition is used. The use of a type definition MUST NOT specify an annotation specified in the type definition.

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

10 Action and Function

10.1 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.

Actions cannot be composed with additional path segments.

An action MAY define zero or more parameters used during the execution of the action.

10.2 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.

Functions MAY be composable.

A function MAY define zero or more parameters used during the execution of the function.

10.3 Name

The action's or function's name MUST be a SimpleIdentifier that MUST be unique within its namespace.

10.4 Bound or Unbound Action or Function

An action or function 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 its 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.

10.5 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 values.

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 SimpleIdentifier of the navigation property to be traversed. A type cast segment names the QualifiedName of the entity type that should be returned from the type cast.

10.6 Action Overloads

Bound actions support overloading by binding parameter type. The combination of action name and the binding parameter type MUST be unique within a namespace.

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

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

10.7 Function Overloads

Bound functions support overloading 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 namespace.
- The combination of function name, binding parameter type, and ordered set of parameter types MUST be unique within a namespace.
- All bound functions with the same function name and binding parameter type within a namespace 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 namespace.
- The combination of function name and ordered set of parameter types MUST be unique within a namespace.
- All unbound functions with the same function name within a namespace 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.

10.8 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.

10.9 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.

10.10 Parameter

An action or function overload MAY specify zero or more parameters.

A bound action or function overload MUST specify at least one parameter, and MUST specify exactly one parameter as its binding parameter.

Each parameter MUST have a name that is a SimpleIdentifier. 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 `Nullable`, `MaxLength`, `Precision`, `Scale`, and `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.

11 Entity Container

Each metadata document used to describe an OData service MUST define exactly one entity container. Entity containers define the entity sets, singletons, function and action imports exposed by the service.

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

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.

11.1 Name

The entity container's name is a SimpleIdentifier that MUST be unique within its namespace.

11.2 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.

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

11.3 Entity Set

Entity sets are top-level collection-valued resources.

An entity set is identified by its name, a SimpleIdentifier 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.

11.4 Singleton

Singletons are top-level single-valued resources.

A singleton is identified by its name, a SimpleIdentifier 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.

11.5 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.

11.5.1 Binding Path

A navigation property binding MUST specify a path to a navigation property of the entity set's or singleton's entity type, one of its subtypes, or a navigation property reached through a chain of containment navigation properties. If the navigation property is defined on a subtype, the path MUST contain the QualifiedName 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 segment MUST be a non-containment navigation property and there MUST NOT be any non-containment navigation properties prior to the final segment.

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.

11.5.2 Binding Target

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

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

If the target is a TargetPath, it MUST resolve to an entity set, singleton, or containment navigation property in scope. The path can traverse containment navigation properties or complex properties before ending in a containment navigation property, but there MUST not be any non-containment navigation properties prior to the final segment.

11.6 Action Import

Action imports sets are top-level resources.

An action import is identified by its name, a SimpleIdentifier 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 SimpleIdentifier or TargetPath value MAY be specified to identify the entity set that contains the returned entities. If a SimpleIdentifier is specified, it MUST resolve to an entity set defined in the same entity container. If a TargetPath is specified, it MUST resolve to an entity set in scope.

11.7 Function Import

Function imports sets are top-level resources.

A function import is identified by its name, a SimpleIdentifier 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 SimpleIdentifier or TargetPath value MAY be specified to identify the entity set that contains the returned entities. If a SimpleIdentifier is specified, it MUST resolve to an entity set defined in the same entity container. If a TargetPath 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.

12 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 can be used to define additional characteristics or capabilities of a metadata element, such as a service, entity type, property, function, action, or parameter. For example, a metadata annotation may define ranges of valid values for a particular property. Metadata annotations are applied in CSDL documents describing or referencing an entity model.

Instance annotations can be used to define additional information associated with a particular result, entity, property, or error; for example, whether a property is read-only for a particular instance. 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. Instance annotations appear in the actual payload as described in **[OData-JSON]**. Annotations that apply across instances should be specified as metadata annotations.

A *vocabulary* is a namespace 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.
- To 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 unknown terms and silently treat unexpected or invalid values (including invalid type, invalid literal expression, etc.) as an unknown value for the term.

12.1 Term

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

12.1.1 Name

The term's name is a SimpleIdentifier that MUST be unique within its namespace.

12.1.2 Type

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

12.1.3 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.

12.1.4 Default Value

A primitive or enumeration term MAY define a default value that is used when the term is applied without providing a value. Whether this short-hand notation is allowed, and how the default values are represented, depends on the CSDL representation.

Default values of type `Edm.String` MUST be represented according to the escaping rules for character data in the CSDL representation. Values of other primitive types MUST be represented according to the

appropriate alternative in the `primitiveValue` rule defined in **[OData-ABNF]**, i.e. `Edm.Binary` as `binaryValue`, `Edm.Boolean` as `booleanValue` etc.

If no value is specified, the default value is `null`.

12.1.5 Applicability

A term MAY specify a list of model constructs it is intended to be applied to. If no list is supplied, the term is not intended to be restricted in its application. As the intended usage may evolve over time, clients SHOULD be prepared for any term to be applied to any 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
ReturnType	Return Type of an Action or Function
Schema	Schema
Singleton	Singleton
Term	Term
TypeDefinition	Type Definition
UrlRef	UrlRef annotation expression

12.1.6 Facets

A term MAY specify values for the Nullable, MaxLength, Precision, Scale, and SRID facets, as well as Unicode for 4.01 and greater payloads. These facets and their implications are described in section 5.3.

12.2 Annotation

An annotation applies a term to a model element and defines how to calculate a value for the term application. Section 12.1.5 specifies which model elements MAY be annotated with a term.

The value of an annotation MAY be a constant expression or dynamic expression. If no expression is specified for a term with a primitive type, the annotation value is the default value of the term. If no expression is specified for a term with a complex type, the annotation value is a complex instance with default values for all its properties. If no expression is specified for a collection-valued term, the annotation evaluates to an empty collection.

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.

12.2.1 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 depends on the CSDL representation.

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 (applies to all overloads)

- Action Import
- Complex Type
- Entity Container
- Entity Set
- Entity Type
- Enumeration Type
- Enumeration Type Member
- Function (applies to all overloads)
- Function Import
- Navigation Property (via type, entity set, or singleton)
- Parameter of an Action or Function (applies to all overloads defining the parameter)
- Property (via type, entity set, or singleton)
- Return Type of an Action or Function (applies to all overloads)
- Singleton
- Term
- 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, in which case the annotation applies to all overloads of the action or function or all parameters of that name across all overloads. External targeting of individual action or function overloads is not possible.

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:

- QualifiedName of schema child
- QualifiedName of schema child followed by a forward slash and name of child element
- QualifiedName of structured type followed by zero or more property, navigation property, or type cast segments, each segment starting with a forward slash
- QualifiedName 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
- QualifiedName of an action or function followed by a forward slash and `$ReturnType`
- QualifiedName of an entity container followed by a segment containing an action or function import name, optionally followed by a forward slash and either a parameter name or `$ReturnType`
- One of the preceding, followed by a forward slash, an at (`@`), the QualifiedName of a term, and optionally a hash (`#`) and the qualifier of an annotation

Example 6: 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.MyFunction
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
```

12.2.2 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 SimpleIdentifier.

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

12.3 Constant Expressions

Constant expressions allow assigning a constant value to an applied term. Representations MUST allow assigning constants of the following types:

- `Edm.Binary`
- `Edm.Boolean`
- `Edm.Byte`
- `Edm.Date`
- `Edm.DateTimeOffset`
- `Edm.Decimal`
- `Edm.Double`
- `Edm.Duration`
- `Edm.Guid`
- `Edm.Int16`
- `Edm.Int32`
- `Edm.Int64`
- `Edm.SByte`
- `Edm.Single`
- `Edm.String`
- `Edm.TimeOfDay`
- Enumeration Types

The concrete syntax is representation-specific.

12.4 Dynamic Expressions

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

12.4.1 Path Expression

The `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 properties of structured types.

The value assigned to the path expression MUST be composed of zero or more path segments joined together by forward slashes (`/`).

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

If a path segment starts with an at (`@`) character, it represents a *term cast*. The at (`@`) character MUST be followed by a QualifiedName that MAY be followed by a hash (`#`) character and a SimpleIdentifier. The QualifiedName preceding the hash character MUST resolve to a term that is in scope, the SimpleIdentifier following the hash sign is interpreted as a Qualifier for the term. If the instance identified by the preceding path part has been annotated with that term (and if present, with that qualifier), the term cast evaluates to the value of that annotation, otherwise it evaluates to the null value. Three special terms are implicitly "annotated" for media entities and stream properties:

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

If a path segment is a SimpleIdentifier, it MUST be the name of a structural property or a navigation property of the instance identified by the preceding path part.

When used within an `Path` expression, a path may contain at most one segment representing a collection-valued structural or navigation property. The result of the expression is the collection of instances resulting from applying the remaining path to each instance in the collection-valued property.

A path may terminate in a `$count` segment if the previous segment is collection-valued, in which case the path evaluates to the number of elements identified by the preceding segment.

If a path segment starts with a navigation property followed by an at (`@`) character, then the at (`@`) character MUST be followed by a QualifiedName that MAY be followed by a hash (`#`) character and a SimpleIdentifier. The QualifiedName preceding the hash character MUST resolve to a term that is in scope, the SimpleIdentifier following the hash sign is interpreted as a Qualifier for the term. If the navigation property has been annotated with that term (and if present, with that qualifier), the path segment evaluates to the value of that annotation, otherwise it evaluates to the null value.

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.

Paths starting with a forward slash (`/`) are evaluated starting at the entity container, and the path part after the first forward slash is interpreted relative to the entity container. Paths not starting with a forward slash are interpreted relative to the annotation target, following the rules specified in the remainder of this section.

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

For annotations embedded within or targeting an entity set or a singleton, the path expression is evaluated starting at the entity set or singleton, i.e. an empty path resolves to the entity set, 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 expression 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 property or navigation property of the type, a type cast, or a term cast.

For annotations embedded within a property of an entity type or complex type, the path expression 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 property of an entity type or complex type, the path expression 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 property 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, or function import, the first segment of a path MUST be a parameter name or `$ReturnType`.

12.4.2 PropertyPath Expression

The `PropertyPath` expression provides a value for terms or term properties that specify one of the built-in abstract types `Edm.PropertyPath`, `Edm.AnyPropertyPath`, or `Edm.AnyPath`. It uses the same syntax and rules as the `Path` expression, with the following exceptions:

- The `PropertyPath` expression may traverse multiple collection-valued structural or navigation properties
- The last path segment MUST resolve either to a structural property in the context of the preceding path part, or to a term cast where the term MUST be of type `Edm.ComplexType`, `Edm.PrimitiveType`, a complex type, an enumeration type, a concrete primitive type, a type definition, or a collection of one of these types.

In contrast to the `Path` expression, the value of the `PropertyPath` expression is the path itself, not the value of the property or the value of the term cast identified by the path.

12.4.3 NavigationPropertyPath Expression

The `NavigationPropertyPath` expression provides a value for terms or term properties that specify the built-in abstract types `Edm.NavigationPropertyPath`, `Edm.AnyPropertyPath`, or `Edm.AnyPath`. It uses the same syntax and rules as the `Path` expression with the following exceptions:

- The `NavigationPropertyPath` expression may traverse multiple collection-valued structural or navigation properties.
- The last path segment MUST resolve to a navigation property in the context of the preceding path part, or to a term cast where the term MUST be of type `Edm.EntityType`, a concrete entity type or a collection of `Edm.EntityType` or concrete entity type.

In contrast to the `Path` expression, the value of the `NavigationPropertyPath` expression is the path itself, not the instance(s) identified by the path.

12.4.4 AnnotationPath Expression

The `AnnotationPath` expression provides a value for terms or term properties that specify the built-in abstract types `Edm.AnnotationPath` or `Edm.AnyPath`. It uses the same syntax and rules as the `Path` expression, with the following exceptions:

- The `AnnotationPath` expression may traverse multiple collection-valued structural or navigation properties.
- The last path segment MUST be a term cast with optional qualifier in the context of the preceding path part.

In contrast to the `Path` expression the value of the `AnnotationPath` 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.

12.4.5 Collection Expression

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.

12.4.6 Record Expression

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.

12.4.7 Conditional Expression

The `If` expression enables a value to be obtained by evaluating a *conditional expression*. It MUST contain exactly three child expressions. There is one exception to this rule: if and only if the `If` expression is an item of a collection expression, the third child expression MAY be omitted (this can be used to conditionally add an element to a collection).

The first child expression is the condition expression 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 element or 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` 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` expression. If no third expression is present, nothing is added to the collection.

12.4.8 Comparison and Logical Operators

Representations MUST allow expressing the following logical and comparison expressions.

They 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

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.

12.4.9 Client-Side Functions

The Apply expression enables a value to be obtained by applying a client-side function. The Apply expression MUST have at least one argument expression. The operand expressions are used as parameters to the client-side function.

12.4.9.1 Function odata.concat

The odata.concat standard 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.

12.4.9.2 Function odata.fillUriTemplate

The odata.fillUriTemplate standard 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 LabeledElement expressions. Each LabeledElement 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 LabeledElement 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 LabeledElement expressions that evaluate to a collection of primitive values.

Key-value maps are represented as LabeledElement 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.

12.4.9.3 Function odata.uriEncode

The odata.uriEncode standard 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 paren-style key syntax.

12.4.10 Cast and IsOf Expressions

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]**.

The `IsOf` 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.

12.4.11 LabeledElement and LabeledElementReference Expressions

The `LabeledElement` expression assigns a name to its single child expression. The value of the child expression can then be reused elsewhere with a `LabeledElementReference` expression.

A `LabeledElement` expression MUST contain exactly one child expression. The value of the child expression is also the value of the `LabeledElement` expression.

A `LabeledElement` expression MUST provide a SimpleIdentifier value as its name that MUST be unique within the schema containing the expression.

The `LabeledElementReference` expression MUST specify the QualifiedName of a `LabeledElement` expression in scope and returns the value of the identified `LabeledElement` expression as its value.

12.4.12 Null Expression

The `Null` expression returns an untyped null value. The null expression may be annotated.

12.4.13 UrlRef Expression

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

The `UrlRef` expression MUST contain exactly one child 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 `UrlRef` 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 `UrlRef` expression. The result of the `UrlRef` expression MUST be type compatible with the type expected by the surrounding expression.

13 Identifiers and Paths

13.1 Namespace

A Namespace is a dot-separated sequence of SimpleIdentifiers with a maximum length of 511 Unicode characters.

13.2 SimpleIdentifier

A SimpleIdentifier is a Unicode character sequence with the following restrictions:

- It consists of at least one and at most 128 Unicode characters.
- 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.

13.3 QualifiedName

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.

13.4 TypeName

The QualifiedName of a built-in primitive or abstract type, a type definition, complex type, enumeration type, or entity type, or a collection of one of these types, see rule `qualifiedTypeName` in **[OData-ABNF]**.

The type must be in scope, i.e. the type MUST be defined in the `Edm` namespace or it MUST be defined in the schema identified by the namespace or alias portion of the qualified name, and the identified schema MUST be defined in the same CSDL document or included from a directly referenced document.

13.5 TargetPath

Target paths are used to refer to other model elements.

The allowed path expressions are:

- The QualifiedName 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 7: Target expressions

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

14 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 model 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 include extended `Edm.Path` expression
4. MUST NOT use `Edm.AnyPath` 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 return the Unicode facet for terms, parameters, and return types
8. MUST NOT include Collections of `Edm.ComplexType` or `Edm.Untyped`
9. MUST NOT specify a key as a property of a related entity
10. SHOULD NOT include new/unknown values for the applicability of a term
11. MAY include new CSDL annotations

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

Appendix A. Acknowledgments

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

Appendix B. Revision History

Revision	Date	Editor	Changes Made
Working Draft 01	2016-10-12	Ralf Handl	Imported content from 4.0 Errata 3 specification and removed XML-specific text
Committee Specification Draft 01	2016-12-08	Michael Pizzo Ralf Handl	Integrated 4.01 features