Serving OData APIs

Table of Contents

Feature Overview

OData is an OASIS standard that enhances plain REST with standardized system query options like `$select` , `$expand` , `$filter` , and others. The following table provides an overview of the feature coverage:

Query Options	Remarks	Node.js	Java
`$search`	Search in multiple/all text elements(1)	✓	✓
`$value`	Retrieves single rows/values	✓	✓
`$top` , `$skip`	Requests paginated results	✓	✓
`$filter`	Like SQL where clause	✓	✓
`$select`	Like SQL select clause	✓	✓
`$orderby`	Like SQL order by clause	✓	✓

Query Options	Remarks	Node.js	Java
`$count`	Gets number of rows for paged results	✓	✓
`$apply`	For data aggregation	✓	✓
`$expand`	Deep-read associated entities	✓	✓
`$compute`	Dynamic expressions for other query options	✓[2]	✓
Lambda Operators	Boolean expressions on a collection	✓	✓ [3]
Parameters Aliases	Replace literal value in URL with parameter alias	✓	✓ [4]

- [1] The elements to be searched are specified with the `@cds.search` annotation.
- [2] Node.js only supports a limited subset in `$select` query option.
- [3] The navigation path identifying the collection can only contain one segment.
- [4] Supported for key values and for parameters of functions only.

System query options can also be applied to an expanded navigation property (nested within `$expand`):

Query Options	Remarks	Node.js	Java
`$select`	Select properties of associated entities	✓	✓
`$filter`	Filter associated entities	✓	✓
`$expand`	Nested expand	✓	✓
`$orderby`	Sort associated entities	✓	✓
`$top` , `$skip`	Paginate associated entities	✓	✓
`$count`	Count associated entities	*n/a*	✓
`$search`	Search associated entities	*n/a*	*n/a*

↳ *Learn more in the **Getting Started guide on odata.org**.*

↳ *Learn more in the tutorials **Take a Deep Dive into OData**.*

Data Modification	Remarks	Node.js	Java
Create an Entity	`POST` request on Entity collection	✓	✓
Update an Entity	`PATCH` or `PUT` request on Entity	✓	✓

Data Modification	Remarks	Node.js	Java
ETags	For avoiding update conflicts	✓	✓
Delete an Entity	`DELETE` request on Entity	✓	✓
Delta Payloads	For nested entity collections in deep updates	*in prog.*	✓
Patch Collection	Update Entity collection with delta	*n/a*	✓ beta

PATCH Entity Collection with Mass Data (Java)

With OData v4, you can update a collection of entities with a *single* PATCH request. The request targets the entity collection in the resource path and provides the request body as a delta payload :

```js
PATCH /CatalogService/Books
Content-Type: application/json

{
  "@context": "#$delta",
  "value": [
    {
      "ID": 17,
      "title": "CAP - what's new in 2023",
      "price": 29.99,
      "author_ID": 999
    },
    {
      "ID": 85,
      "price": 9.99
    },
    {
      "ID": 42,
      "@removed": { "reason": "deleted" }
    }
  ]
}
```

The system executes PATCH requests with a delta payload using batch delete and upsert statements. These requests are more efficient than OData batch requests .

Use PATCH on entity collections to upload mass data using a dedicated service secured with role-based authorization. Enable delta updates explicitly by annotating the entity with

```cds
@Capabilities.UpdateRestrictions.DeltaUpdateSupported
```

Limitations:

- Conflict detection via ETags is not supported.
- The system bypasses draft flow. `IsActiveEntity` must be `true` .
- The system ignores draft locks. Active entities are updated or deleted without canceling drafts.
- Added and deleted links are not supported.
- The header `Prefer=representation` is not yet supported.
- The `continue-on-error` preference is not yet supported.
- The generic CAP handler support for upsert is limited, for example, audit logging is not supported.

Mapping of CDS Types

The following table lists CDS's built-in types and their mapping to the OData EDM type system.

CDS Type	OData V4
`UUID`	*Edm.Guid* (1)
`Boolean`	*Edm.Boolean*
`UInt8`	*Edm.Byte*

CDS Type	OData V4
`Int16`	*Edm.Int16*
`Int32`	*Edm.Int32*
`Integer`	*Edm.Int32*
`Int64`	*Edm.Int64*
`Integer64`	*Edm.Int64*
`Decimal`	*Edm.Decimal*
`Double`	*Edm.Double*
`Date`	*Edm.Date*
`Time`	*Edm.TimeOfDay*
`DateTime`	*Edm.DateTimeOffset*
`Timestamp`	*Edm.DateTimeOffset* with Precision="7"
`String`	*Edm.String*
`Binary`	*Edm.Binary*
`LargeBinary`	*Edm.Binary*
`LargeString`	*Edm.String*
`Map`	represented as an empty, open complex type
`Vector`	not supported (2)

> **(1)** Mapping can be changed with, for example, `@odata.Type='Edm.String'`

> **(2)** Type `cds.Vector` must not appear in an OData service

OData V2 has the following differences:

CDS Type	OData V2
`Date`	*Edm.DateTime* with `sap:display-format="Date"`
`Time`	*Edm.Time*
`Map`	not supported

Overriding Type Mapping

Use the annotation `@odata.Type` first to override standard type mappings, then additionally define `@odata {MaxLength, Precision, Scale, SRID}`.

`@odata.Type` is effective on scalar CDS types only and the value must be a valid OData (EDM) primitive type for the specified protocol version. Unknown types and non-matching facets are silently ignored. No further value constraint checks are applied.

These annotations allow you to produce additional OData EDM types that are not available in the standard type mapping. Use this approach during the import of external service APIs. See Using Services.

```cds
entity Foo {
  // ...
  @odata: { Type: 'Edm.GeometryPolygon', SRID: 0 }
  geoCollection : LargeBinary;
};
```

Another prominent use case is the CDS type `UUID`, which maps to `Edm.Guid` by default. However, the OData standard imposes restrictive rules for *Edm.Guid* values. For example, only hyphenated strings are allowed, which can conflict with existing data. You can override the default mapping as follows:

```cds
entity Books {
  key ID : UUID @odata.Type:'Edm.String';
  // ...
}
```

WARNING

This annotation affects the client-side facing API only. No automatic data modification occurs behind the scenes, such as rounding, truncation, or conversion. You must perform all the required modifications on the data stream so that the values match their type in the API. If you don't do the required conversions, you can "cast" any scalar CDS type into any incompatible EDM type:

```cds
entity Foo {
  // ...
  @odata: {Type: 'Edm.Decimal', Scale: 'floating' }
  str: String(17) default '17.4';
}
```

This translates into the following OData API contract:

```xml
<Property Name="str" Type="Edm.Decimal" Scale="floating" DefaultValue="17.4",
```

The client can now rightfully expect float numbers to be transmitted, but in reality the values are still strings.

OData Annotations

The following sections explain how to add OData annotations to CDS models and how to map them to EDMX outputs. The translation considers only annotations defined in the vocabularies mentioned in Annotation Vocabularies.

Terms and Properties

OData defines a strict two-fold key structure composed of `@<Vocabulary>.<Term>` . All annotations are always specified as a *Term* with either a primitive value, a record value, or collection values. The properties themselves may, in turn, be primitives, records, or collections.

Example

```cds
@Common.Label: 'Customer'
@UI.HeaderInfo: {
  TypeName       : 'Customer',
  TypeNamePlural : 'Customers',
  Title          : { Value : name }
}
entity Customers { /* ... */ }
```

This is represented in CSN as follows:

```jsonc
{"definitions":{
  "Customers":{
    "kind": "entity",
    "@Common.Label": "Customer",
    "@UI.HeaderInfo.TypeName": "Customer",
    "@UI.HeaderInfo.TypeNamePlural": "Customers",
    "@UI.HeaderInfo.Title.Value": {"=": "name"},
    /* ... */
  }
}}
```

And would render to EDMX as follows:

```xml
<Annotations Target="MyService.Customers">
  <Annotation Term="Common.Label" String="Customer"/>
  <Annotation Term="UI.HeaderInfo">
    <Record Type="UI.HeaderInfoType">
      <PropertyValue Property="TypeName" String="Customer"/>
      <PropertyValue Property="TypeNamePlural" String="Customers"/>
      <PropertyValue Property="Title">
        <Record Type="UI.DataField">
          <PropertyValue Property="Value" Path="name"/>
        </Record>
      </PropertyValue>
    </Record>
  </Annotation>
</Annotations>
```

TIP

The value for `@UI.HeaderInfo` is flattened to individual key-value pairs in CSN and 'restructured' to a record for OData exposure in EDMX.

For each annotated target definition in CSN, the rules for restructuring from CSN sources are:

1. Annotations with a single-identifier key are skipped (as OData annotations always have a `@Vocabulary.Term...` key signature).
2. All individual annotations with the same `@<Vocabulary.Term>` prefix are collected.
3. If there's only one annotation without a suffix, → that one is a scalar or array value of an OData term.
4. If there are more annotations with suffix key parts →, it's a record value for the OData term.

Qualified Annotations

OData provides qualified annotations, which allow you to specify different values for a given property. CDS syntax for annotations was extended to allow appending OData-style qualifiers after a `#` sign to an annotation key, but always only as the last component of a key in the syntax.

For example, this is supported:

```cds
@Common.Label: 'Customer'
@Common.Label#Legal: 'Client'
@Common.Label#Healthcare: 'Patient'
@Common.ValueList: {
  Label: 'Customers',
  CollectionPath:'Customers'
}
@Common.ValueList#Legal: {
  Label: 'Clients',
  CollectionPath:'Clients'
}
```

and would render as follows in CSN:

```json
{
  "@Common.Label": "Customer",
  "@Common.Label#Legal": "Clients",
  "@Common.Label#Healthcare": "Patients",
  "@Common.ValueList.Label": "Customers",
  "@Common.ValueList.CollectionPath": "Customers",
  "@Common.ValueList#Legal.Label": "Clients",
  "@Common.ValueList#Legal.CollectionPath": "Clients",
}
```

CDS provides no interpretation and no special handling for these qualifiers. You must write and apply them exactly as your chosen OData vocabularies specify them.

Primitives

> Note: The `@Some` annotation isn't a valid term definition. The following example illustrates the rendering of primitive values.

The system maps primitive annotation values (Strings, Numbers, `true` , and `false`) to corresponding OData annotations as follows:

```cds
@Some.Boolean: true
@Some.Integer: 1
@Some.Number: 3.14
@Some.String: 'foo'
```

```xml
<Annotation Term="Some.Boolean" Bool="true"/>
<Annotation Term="Some.Integer" Int="1"/>
<Annotation Term="Some.Number" Decimal="3.14"/>
<Annotation Term="Some.String" String="foo"/>
```

Null Value

A `null` value can be set either as an annotation expression or as a dynamic expression:

```cds
@Some.NullXpr:  (null)                  // annotation expression, short form
@Some.NullFunc: ($Null())               // annotation expression, functional
@Some.NullDyn:  { $edmJson: { $Null } } // dynamic expression
```

All three expressions result in the following rendering:

```xml
<Annotation Term="Some.Null">
  <Null/>
</Annotation>
```

↳ *Have a look at our **CAP SFLIGHT** sample, showcasing the usage of OData annotations.*

Records

> Note: The `@Some` annotation isn't a valid term definition. The following example illustrates the rendering of record values.

The system maps record-like source structures to `<Record>` nodes in EDMX, with primitive types translated analogously to what was mentioned earlier:

cds

```cds
@Some.Record: {
  Null: (null),
  Boolean: true,
  Integer: 1,
  Number: 3.14,
  String: 'foo'
}
```

xml

```xml
<Annotation Term="Some.Record">
  <Record>
    <PropertyValue Property="Null"><Null/></PropertyValue>
    <PropertyValue Property="Boolean" Bool="true"/>
    <PropertyValue Property="Integer" Int="1"/>
    <PropertyValue Property="Number" Decimal="3.14"/>
    <PropertyValue Property="String" String="foo"/>
  </Record>
</Annotation>
```

If possible, the type of the record in OData is deduced from the information in the OData Annotation Vocabularies:

cds

```cds
@Common.ValueList: {
  CollectionPath: 'Customers'
}
```

xml

```xml
<Annotation Term="Common.ValueList">
  <Record Type="Common.ValueListType">
    <PropertyValue Property="CollectionPath" String="Customers"/>
  </Record>
</Annotation>
```

Frequently, the OData record type cannot be determined unambiguously, for example if the type found in the vocabulary is abstract. Then you need to explicitly specify the type by adding a property named `$Type` in the record. For example:

cds

```cds
@UI.Facets : [{
  $Type   : 'UI.CollectionFacet',
  ID      : 'Customers'
}]
```

xml

```xml
<Annotation Term="UI.Facets">
  <Collection>
    <Record Type="UI.CollectionFacet">
      <PropertyValue Property="ID" String="Travel"/>
    </Record>
  </Collection>
</Annotation>
```

There is one exception for a very prominent case: if the deduced record type is `UI.DataFieldAbstract`, the compiler by default automatically chooses `UI.DataField`:

cds

```cds
@UI.Identification: [{
  Value: deliveryId
}]
```

xml

```xml
<Annotation Term="UI.Identification">
  <Collection>
    <Record Type="UI.DataField">
      <PropertyValue Property="Value" Path="deliveryId"/>
    </Record>
  </Collection>
</Annotation>
```

To overwrite the default, use an explicit `$Type` like shown previously.

↳ *Have a look at our **CAP SFLIGHT** sample, showcasing the usage of OData annotations.*

Collections

> Note: The `@Some` annotation isn't a valid term definition. The following example illustrates the rendering of collection values.

The system maps arrays to `<Collection>` nodes in EDMX. If primitives appear as direct elements of the array, these elements are wrapped into individual primitive child nodes of the resulting collection as is. The system applies the rules for records and collections recursively:

cds

```cds
@Some.Collection: [
  null, true, 1, 3.14, 'foo',
  { $Type:'UI.DataField', Label:'Whatever', Hidden }
]
```

```xml
<Annotation Term="Some.Collection">
  <Collection>
    <Null/>
    <Bool>true</Bool>
    <Int>1</Int>
    <Decimal>3.14</Decimal>
    <String>foo</String>
    <Record Type="UI.DataField">
      <PropertyValue Property="Label" String="Whatever"/>
      <PropertyValue Property="Hidden" Bool="True"/>
    </Record>
  </Collection>
</Annotation>
```

References

> Note: The `@Some` annotation isn't a valid term definition. The following example illustrates the rendering of reference values.

The system maps references in CDS annotations to `Path` properties or nested `<Path>` elements, respectively:

```cds
@Some.Term: My.Reference
@Some.Record: {
  Value: My.Reference
}
@Some.Collection: [
  My.Reference
]
```

```xml
<Annotation Term="Some.Term" Path="My/Reference"/>
<Annotation Term="Some.Record">
  <Record>
    <PropertyValue Property="Value" Path="My/Reference"/>
  </Record>
</Annotation>
<Annotation Term="Some.Collection">
  <Collection>
```

```
    <Path>My/Reference</Path>
  </Collection>
</Annotation>
```

As the compiler isn't aware of the semantics of such references, the mapping is very simplistic: each `.` in a path is replaced by a `/`. Use expression-valued annotations for more convenience.

Use a dynamic expression if the generic mapping can't produce the desired `<Path>`:

cds

```
@Some.Term: {$edmJson: {$Path: '/com.sap.foo.EntityContainer/EntityName/FieldN
```

xml

```
<Annotation Term="Some.Term">
  <Path>/com.sap.foo.EntityContainer/EntityName/FieldName</Path>
</Annotation>
```

Enumeration Values

The system maps enumeration symbols to corresponding `EnumMember` properties in OData.

Here are a couple of examples of enumeration values and the annotations that are generated. The first example is for a term in the Common vocabulary :

cds

```
@Common.TextFormat: #html
```

xml

```
<Annotation Term="Common.TextFormat" EnumMember="Common.TextFormatType/html"/:
```

The second example is for a (record type) term in the Communication vocabulary :

cds

```
@Communication.Contact: {
  gender: #F
}
```

xml

```
<Annotation Term="Communication.Contact">
  <Record Type="Communication.ContactType">
    <PropertyValue Property="gender" EnumMember="Communication.GenderType/F"/:
```

```
  </Record>
</Annotation>
```

Expressions

If the value of an OData annotation is an expression, the OData backend provides improved handling of references and automatic mapping from CDS expression syntax to OData expression syntax.

One of the main use cases for such dynamic expressions is SAP Fiori. Examples:

```cds
@UI.Hidden: (status <> 'visible')
@UI.CreateHidden : (to_Travel.TravelStatus.code != #Open)
```

Note that SAP Fiori supports dynamic expressions only for specific annotations .

Flattening

In contrast to simple references, the references in expression-like annotation values are correctly handled during model transformations, like other references in the model. When the CDS model is flattened for OData, the flattening is consequentially also applied to these references, and they are translated to the flat model.

> **TIP**
>
> Although CAP supports structured types and elements, we recommend using them only if they bring a real benefit. In general, you should keep your models as flat as possible.

Example:

```cds
type Price {
  @Measures.ISOCurrency: (currency)
  amount : Decimal;
  currency : String(3);
}

service S {
  entity Product {
    key id : Integer;
    name : String;
    price : Price;
  }
}
```

Structured element *price* of *S.Product* is unfolded to flat elements *price_amount* and *price_currency* . Accordingly, the reference in the annotation is rewritten from *currency* to *price_currency* :

```xml
<Schema Namespace="S">
  <!-- ... -->
  <EntityType Name="Product">
    <!-- ... -->
    <Property Name="price_amount" Type="Edm.Decimal" Scale="variable"/>
    <Property Name="price_currency" Type="Edm.String" MaxLength="3"/>
  </EntityType>
  <Annotations Target="S.Product/price_amount">
    <Annotation Term="Measures.ISOCurrency" Path="price_currency"/>
  </Annotations>
</Schema>
```

Example:

```cds
service S {
  entity E {
    key id : Integer;
    f : Association to F;
    @Some.Term: (f.struc.y)
    val : Integer;
  }
  entity F {
    key id : Integer;
    struc {
      y : Integer;
    }
  }
}
```

The OData backend is aware of the semantics of a path and distinguishes association path steps from structure access. The CDS path *f.struc.y* is translated to the OData path *f/struc_y* :

```xml
<Schema Namespace="S">
  <!-- ... -->
  <EntityType Name="E">
    <!-- ... -->
    <NavigationProperty Name="f" Type="S.F"/>
    <Property Name="val" Type="Edm.Int32"/>
  </EntityType>
  <EntityType Name="F">
    <!-- ... -->
    <Property Name="struc_y" Type="Edm.Int32"/>
  </EntityType>
  <Annotations Target="S.E/val">
    <Annotation Term="Some.Term" Path="f/struc_y"/>
  </Annotations>
</Schema>
```

Managed Associations

The OData backend translates managed associations into unmanaged associations plus explicit foreign key elements. During this translation, the system copies annotations assigned to the managed association to the respective foreign key elements.

Example:

```cds
service S {
  entity Authors { key ID : Integer; name : String; }
  entity Books   { key ID : Integer; author : Association to Authors; }

  annotate Books:author with @Common.Text: (author.name);
}
```

Resulting OData API:

```xml
<Schema Namespace="S">
  <!-- ... -->
  <EntityType Name="Authors">
    <!-- ... -->
    <Property Name="name" Type="Edm.String"/>
  </EntityType>
  <EntityType Name="Books">
    <!-- ... -->
```

```xml
      <NavigationProperty Name="author" Type="S.Authors"/>
      <Property Name="author_ID" Type="Edm.Int32"/>
    </EntityType>
    <Annotations Target="S.Books/author_ID">
      <Annotation Term="Common.Text" Path="author/name"/>
    </Annotations>
  </Schema>
```

Instead of relying on this copy mechanism, you can also explicitly annotate a foreign key element:

```cds
annotate Books:author.ID with @Common.Text: ($self.author.name);  // here $se
```

The system always rewrites a path that addresses a key element in the target of a managed association to address the local foreign key element.

Example:

```cds
service S {
  entity Travels {
    key id : Integer;
    status : Association to TravelStatus;
  };
  entity TravelStatus {
    key code : String(1) enum {Open = 'O'; Accepted = 'A'; Canceled = 'X'; };
  }
  @UI.CreateHidden : (travel.status.code != #Open)
  entity Bookings {
    key id : Integer;
    travel : Association to Travels;
  }
}
```

Resulting OData API:

```xml
<Schema Namespace="S">
  <!-- ... -->
  <EntityType Name="Travels">
    <!-- ... -->
    <NavigationProperty Name="status" Type="S.TravelStatus"/>
```

```
      <Property Name="status_code" Type="Edm.String" MaxLength="1"/>
    </EntityType>
    <EntityType Name="TravelStatus">
      <!-- ... -->
    </EntityType>
    <EntityType Name="Bookings">
      <!-- ... -->
      <NavigationProperty Name="travel" Type="S.Travels"/>
    </EntityType>
    <Annotations Target="S.Bookings">
      <Annotation Term="UI.CreateHidden">
        <Ne>
          <Path>travel/status_code</Path>
          <String>O</String>
        </Ne>
      </Annotation>
    </Annotations>
  </Schema>
```

Expression Translation

If the expression you provide as an annotation value is more complex than just a reference, the OData backend translates CDS expressions to the corresponding OData expression syntax. The backend rejects expressions that are not applicable in an OData API.

> **INFO**
>
> While the flattening of references described in the preceding section is applied to all annotations, the syntactic translation of expressions is only done for annotations defined in one of the OData vocabularies.

The following operators and clauses of CDL are supported:

- `case when ... then ... else ...` and the logical ternary operator `? :`
- Logical: `and` , `or` , `not`
- Relational: `=` , `<>` , `!=` , `<` , `<=` , `>` , `>=` , `in` , `between ... and ...`
- Unary `+` and `-`
- Arithmetic: `+` , `-` , `*` , `/`
- Concat: `||`

- *cast(...)*

Example:

```cds
@Some.Xpr: ( -(a + b) )
```

```xml
<Annotation Term="Some.Xpr">
  <Neg>
    <Add>
      <Path>a</Path>
      <Path>b</Path>
    </Add>
  </Neg>
</Annotation>
```

You can use such expressions, for example, for some Fiori UI annotations :

```cds
service S {
  @UI.LineItem: [ // ...
  {
    Value: (status),
    Criticality: ( status = 'O' ? 2 : ( status = 'A' ? 3 : 0 ) )
  }]
  entity Order {
    key id : Integer;
    // ...
    status : String;
  }
}
```

If you need to access an element of an entity in an annotation for a bound action or function, use a path that navigates via an explicitly defined binding parameter.

Example:

```cds
service S {
  entity Order {
    key id : Integer;
    // ...
    status : String;
  } actions {
    @Core.OperationAvailable: ( :in.status <> 'A' )
    action accept (in: $self)
  }
}
```

In addition, the following functions are supported:

- `$Null()` representing the `null` value [`Null`](annotation expression).
- `Div(...)` (or `$Div(...)`) and `Mod(...)` (or `$Mod(...)`) for integer division and modulo
- `Has(...)` (or `$Has(...)`)
- the functions listed in sections 5.1.1.5 through 5.1.1.11 of OData URL conventions
    - See examples below for the syntax for `cast` and `isof` (section 5.1.1.10)
    - The names of the geo functions (section 5.1.1.11) need to be escaped like `![geo.distance]`
- `fillUriTemplate(...)` and `uriEncode(...)`
- `Type(...)` (or `$Type(...)`) is to be used to specify a type name with their corresponding type facets such as `MaxLength(...)` , `Precision(...)` , `Scale(...)` and `SRID(...)` (or `$MaxLength(...)` , `$Precision(...)` , `$Scale(...)` , `$SRID(...)`)

Example:

```cds
@Some.Func1: ( concat(a, b, c) )
@Some.Func2: ( round(aNumber) )
@Some.Func3: ( $Cast(aValue, $Type('Edm.Decimal', $Precision(38), $Scale(19))
@Some.Func4: ( $IsOf(aValue, $Type('Edm.Decimal', $Precision(38), $Scale(19))
@Some.Func5: ( ![geo.distance](a, b) )
@Some.Func6: ( fillUriTemplate(a, b) )
```

If a functional expression starts with a `$` , all inner function must also be `$` functions and vice versa. Instead of `[$]Type(...)` an EDM primitive type name can be directly used as function name like in CDL.

It is worth to mention that there are two alternatives for the cast function, one in the EDM and one in the CDS domain:

```cds
@Some.ODataStyleCast:  ( Cast(aValue, Decimal(38, 'variable') ) )  // => Edm.I
@Some.ODataStyleCast2: ( Cast(aValue, PrimitiveType()) )           // => Edm.I
@Some.SQLStyleCast:    ( cast(aValue as Decimal(38, variable)) )   // => cds.I
@Some.SQLStyleCast2:   ( cast(aValue as String) )                  // => cds.S
```

Both `cast` functions look similar, but there are some differences:

The OData style `Cast` *function* starts with a capital letter and the SQL `cast` *operator* uses the keyword `as` to delimit the element reference from the type specifier. The OData `Cast` requires an EDM primitive type to be used either as `[$]Type()` or as direct type function whereas the SQL `cast` requires a scalar CDS type as argument which is then converted into the corresponding EDM primitive type.

INFO

CAP only provides a syntactic translation. It is up to each client whether an expression value is supported for a particular annotation. See, for example, SAP Fiori Elements' list of supported annotations .

Use a dynamic expression if the desired EDMX expression cannot be obtained via the automatic translation of a CDS expression.

Annotating Annotations

OData can annotate annotations. This often occurs in combination with enums like `UI.Importance` and `UI.TextArrangement` . CDS has no corresponding language feature. For OData annotations, you can achieve nesting in the following way:

- To annotate a Record, add an additional element to the CDS source structure. The name of this element is the full name of the annotation, including the `@` . See `@UI.Importance` in the following example.
- To annotate a single value or a Collection, add a parallel annotation that has the nested annotation name appended to the outer annotation name. See `@UI.Criticality` and `@UI.TextArrangement` in the following example.

```cds
@UI.LineItem: [
  {Value: ApplicationName, @UI.Importance: #High},
  {Value: Description},
  {Value: SourceName},
  {Value: ChangedBy},
  {Value: ChangedAt}
]
@UI.LineItem.@UI.Criticality: #Positive


@Common.Text: Text
@Common.Text.@UI.TextArrangement: #TextOnly
```

Alternatively, annotating a single value or a Collection by turning them into a structure with an artificial property `$value` is still possible, but deprecated:

cds

```
  @UI.LineItem: {
    $value:[ /* ... */ ], @UI.Criticality: #Positive
  }

  @Common.Text: {
    $value: Text, @UI.TextArrangement: #TextOnly
  }
```

As `TextArrangement` is common, there's a shortcut for this specific situation:

cds

```
  ...
  @Common: {
    Text: Text, TextArrangement: #TextOnly
  }
```

In any case, the resulting EDMX is:

xml

```
<Annotation Term="UI.LineItem">
  <Collection>
    <Record Type="UI.DataField">
      <PropertyValue Property="Value" Path="ApplicationName"/>
      <Annotation Term="UI.Importance" EnumMember="UI.ImportanceType/High"/>
    </Record>
    ...
  </Collection>
  <Annotation Term="UI.Criticality" EnumMember="UI.CriticalityType/Positive"/
</Annotation>
<Annotation Term="Common.Text" Path="Text">
```

```xml
    <Annotation Term="UI.TextArrangement" EnumMember="UI.TextArrangementType/Te
  </Annotation>
```

EDM JSON Expression Syntax

> **Use CDS expression syntax**
>
> Use the EDM JSON expression syntax only as a fallback mechanism. Whenever possible, use expression-like annotation values instead. For the following example, simply write `@UI.Hidden: (status <> 'visible')`.

In case you want to have an expression as value for an OData annotation that cannot be written as a CDS expression , you can use the "edm-json inline mechanism" by providing an EDM JSON expression as defined in the JSON representation of the OData Common Schema Language enclosed in `{ $edmJson: { ... }}`.

Note that here the CDS syntax for string literals with single quotes (`'foo'`) applies, and that paths are not automatically recognized but need to be written as `{$Path: 'fieldName'}`. The CDS compiler translates the expression into the corresponding XML representation .

For example, the CDS annotation:

```cds
@UI.Hidden: {$edmJson: {$Ne: [{$Path: 'status'}, 'visible']}}
```

is translated to:

```xml
<Annotation Term="UI.Hidden">
  <Ne>
    <Path>status</Path>
    <String>visible</String>
  </Ne>
</Annotation>
```

sap: Annotations

In general, back ends and SAP Fiori UIs understand or expect OData V4 annotations. You should use those rather than the OData V2 SAP extensions.

If necessary, CDS automatically translates OData V4 annotations to OData V2 SAP extensions when you invoke it with `v2` as the OData version. This means you shouldn't need to deal with this at all.

Nevertheless, in case you need to do so, you can add `sap:...` attribute-style annotations as follows:

```cds
@sap.applicable.path: 'to_eventStatus/EditEnabled'
action EditEvent(...) returns SomeType;
```

Which would render to OData EDMX as follows:

```xml
<FunctionImport Name="EditEvent" ...
  sap:applicable-path="to_eventStatus/EditEnabled">
  ...
</FunctionImport>
```

The rules are:

- Only strings are supported as values.
- The first dot in `@sap.` is replaced by a colon `:`.
- Subsequent dots are replaced by dashes.

Differences to ABAP

In contrast to ABAP CDS, we apply a **generic, isomorphic approach** where names and positions of annotations are exactly as specified in the OData Vocabularies. This has the following advantages:

- Single source of truth — users only need to consult the official OData specs
- Speed — we don't need complex case-by-case mapping logic
- No bottlenecks — we always support the full set of OData annotations
- Bidirectional mapping — we can translate CDS to EDMX and vice versa

Last but not least, it also saves us lots of effort as we don't have to write derivatives of all the OData vocabulary specs.

Annotation Vocabularies

When translating a CDS model to an OData API, by default only those annotations are considered that are part of the standard OASIS or SAP vocabularies listed below. You can add further vocabularies to the translation process using configuration.

OASIS Vocabularies

Vocabulary	Description
@Aggregation	for describing aggregatable data
@Authorization	for authorization requirements
@Capabilities	for restricting capabilities of a service
@Core	for general purpose annotations
@JSON	for JSON properties
@Measures	for monetary amounts and measured quantities
@Repeatability	for repeatable requests
@Temporal	for temporal annotations
@Validation	for adding validation rules

SAP Vocabularies

Vocabulary	Description
@Analytics	for annotating analytical resources
@CodeList	for code lists
@Common	for all SAP vocabularies
@Communication	for annotating communication-relevant information
@DataIntegration	for data integration
@Hierarchy	for hierarchies
@PDF	for PDF

Vocabulary	Description
@PersonalData	for annotating personal data
@Session	for sticky sessions for data modification
@UI	for presenting data in user interfaces

↳ *Learn more about annotations in CDS and OData and how they work together*

Additional Vocabularies

Assuming you have a vocabulary `com.MyCompany.vocabularies.MyVocabulary.v1` , you can set the configuration option `cds.cdsc.odataVocabularies.MyVocabulary` ☼ .

With this configuration, all annotations prefixed with `MyVocabulary` are considered in the translation.

```cds
service S {
  @MyVocabulary.MyAnno: 'My new Annotation'
  entity E { /*...*/ };
};
```

The annotation is added to the OData API, as well as the mandatory reference to the vocabulary definition:

```xml
<edmx:Reference Uri="link to vocabulary document">
  <edmx:Include Alias="MyVocabulary" Namespace="com.MyCompany.vocabularies.My
</edmx:Reference>
...
<Annotations Target="S.E">
  <Annotation Term="MyVocabulary.MyAnno" String="My new Annotation"/>
</Annotations>
```

The compiler evaluates neither annotation values nor the URI. It is your responsibility to make the URI accessible if necessary. Unlike for the standard vocabularies listed above, the compiler has no access to the content of the vocabulary, so the values are translated generically.

Data Aggregation

Data aggregation in OData V4 is leveraged by the `$apply` system query option, which defines a pipeline of transformations that is applied to the *input set* specified by the URI. On the *result set* of the pipeline, the standard system query options come into effect.

Example

```http
GET /Orders(10)/books?
    $apply=filter(year eq 2000)/
           groupby((author/name),aggregate(price with average as avg))/
    orderby(title)/
    top(3)
```

This request operates on the books of the order with ID 10. First, it filters out the books from the year 2000 to create an intermediate result set. The system then groups the intermediate result set by author name and averages the price. Finally, the system sorts the result set by title and retains only the top three entries.

WARNING

If the `groupby` transformation only includes a subset of the entity keys, the result order might be unstable.

Transformations

Transformation	Description	Node.js	Java
`filter`	filter by filter expression	✓	✓
`search`	filter by search term or expression	*n/a*	✓
`groupby`	group by dimensions and aggregates values	✓	✓
`aggregate`	aggregate values	✓	✓
`compute`	add computed properties to the result set	*n/a*	✓

Transformation	Description	Node.js	Java
`expand`	expand navigation properties	*n/a*	*n/a*
`concat`	append additional aggregation to the result	✓	✓
`skip` / *`top`*	paginate	✓	✓
`orderby`	sort the input set	✓	✓
`topcount` / *`bottomcount`*	retain highest/lowest *n* values	*n/a*	*n/a*
`toppercent` / *`bottompercent`*	retain highest/lowest *p*% values	*n/a*	*n/a*
`topsum` / *`bottomsum`*	retain *n* values limited by sum	*n/a*	*n/a*
`TopLevels`	retain only *n* levels of a hierarchy	✓[2]	✓[1,2]
`ancestors/descendants`	retain ancestors/descendants of specific nodes	✓[2]	✓[1,2]

[1] - supported on SAP HANA, H2 ad PostgreSQL only [2] - only to support requests from the UI5 Tree Table

`concat`

The *`concat`* transformation applies additional transformation sequences to the input set and concatenates the result:

```http
GET /Books?$apply=
    filter(author/name eq 'Bram Stroker')/
    concat(
        aggregate($count as totalCount),
        groupby((year), aggregate($count as countPerYear)))
```

This request filters all books, keeping only books by Bram Stroker. From these books, *`concat`* calculates (1) the total count of books *and* (2) the count of books per year. The result is heterogeneous.

The *`concat`* transformation must be the last of the apply pipeline. If *`concat`* is used, then *`$apply`* can't be used in combination with other system query options.

`skip` , *`top`* , and *`orderby`*

Beyond the standard transformations specified by OData, CDS Java supports the transformations `skip` , `top` , and `orderby` that allow you to sort and paginate an input set:

```http
GET /Order(10)/books?
    $apply=orderby(price desc)/
           top(500)/
           groupby((author/name),aggregate(price with max as maxPrice))
```

This query groups the 500 most expensive books by author name and determines the price of the most expensive book per author.

Aggregation Methods

Aggregation Method	Description	Node.js	Java
`min`	smallest value	✓	✓
`max`	largest	✓	✓
`sum`	sum of values	✓	✓
`average`	average of values	✓	✓
`countdistinct`	count of distinct values	✓	✓
custom method	custom aggregation method	*n/a*	*n/a*
custom aggregate	predefined custom aggregate	✓	✓
`$count`	number of instances in input set	✓	✓

Custom Aggregates

Instead of explicitly using an expression with an aggregation method in the `aggregate` transformation, the client can use a *custom aggregate*. A custom aggregate can be considered as a virtual property that aggregates the input set. It's calculated on the server side. The client doesn't know *How* the custom aggregate is calculated.

They can only be used for the special case when a default aggregation method can be specified declaratively on the server side for a measure.

A custom aggregate is declared in the CDS model as follows:

- The measure must be annotated with an `@Aggregation.default` annotation that specifies the aggregation method.
- The CDS entity should be annotated with an `@Aggregation.CustomAggregate` annotation to expose the custom aggregate to the client.

```cds
@Aggregation.CustomAggregate#stock : 'Edm.Decimal'
entity Books as projection on bookshop.Books {
  ID,
  title,

  @Aggregation.default: #SUM
  stock
};
```

With this definition, it's now possible to use the custom aggregate `stock` in an `aggregate` transformation:

```http
GET /Books?$apply=aggregate(stock) HTTP/1.1
```

which is equivalent to:

```http
GET /Books?$apply=aggregate(stock with sum as stock) HTTP/1.1
```

Currencies and Units of Measure

If a property represents a monetary amount, it may have a related property that indicates the amount's *currency code*. Analogously, a property representing a measured quantity can be related to a *unit of measure*. To indicate that a property is a currency code or a unit of measure, it can be annotated with the Semantics Annotations `@Semantics.currencyCode` or `@Semantics.unitOfMeasure`. The aggregation method (typically, sum) is specified with the `@Aggregation.default` annotation.

```cds
@Aggregation.CustomAggregate#amount   : 'Edm.Decimal'
@Aggregation.CustomAggregate#currency : 'Edm.String'
entity Sales {
  key id        : GUID;
      productId : GUID;
      @Semantics.amount.currencyCode: 'currency'
      @Aggregation.default: #SUM
      amount    : Decimal(10,2);
      @Semantics.currencyCode
```

```
        currency  : String(3);
    }
```

All properties annotated with `@Semantics.currencyCode` or `@Semantics.unitOfMeasure` are exposed as a custom aggregate with the property's name that returns:

- The property's value if it's unique within a group of dimensions
- `null` otherwise

A custom aggregate for a currency code or unit of measure should also be exposed by the `@Aggregation.CustomAggregate` annotation. Moreover, a property for a monetary amount or a measured quantity should be annotated with `@Semantics.amount.currencyCode` or `@Semantics.quantity.unitOfMeasure` to reference the corresponding property that holds the amount's currency code or the quantity's unit of measure, respectively.

Other Features

Feature	Node.js	Java
use path expressions in transformations	✓	✓
chain transformations	✓	✓
chain transformations within group by	*n/a*	*n/a*
`groupby` with `rollup` / `$all`	*n/a*	*n/a*
`$expand` result set of `$apply`	*n/a*	✓
`$filter` / `$search` result set	✓	✓
sort result set with `$orderby`	✓	✓
paginate result set with `$top` / `$skip`	✓	✓

Open Types

An entity type or a complex type may be declared as *open*, which allows clients to add properties dynamically to instances of the type. Clients do this by specifying uniquely named property values in the payload used to insert or update an instance of the type. To indicate that the entity or complex type is open, annotate the corresponding type with `@open`:

```cds
service CatalogService {
  @open
  entity Book {
    key id : Integer;
  }
}
```

The *cds build* for OData v4 renders the entity type `Book` in `edmx` with the `OpenType` attribute set to `true`:

```xml
<EntityType Name="Book" OpenType="true"> // [!code focus]
  <Key>
    <PropertyRef Name="id"/>
  </Key>
  <Property Name="id" Type="Edm.Integer" Nullable="false"/>
</EntityType>
```

The entity `Book` is open, which allows the client to enrich the entity with additional properties.

Example 1:

```json
{"id": 1, "title": "Tow Sawyer"}
```

Example 2:

```json
{"title": "Tow Sawyer",
 "author": { "name": "Mark Twain", "age": 74 } }
```

Open types can also be referenced in non-open types and entities. This, however, doesn't make the referencing entity or type open.

```cds
service CatalogService {
  type Order {
    guid: Integer;
    book: Book;
    }

    @open
    type Book {}
  }
```

The following payload for `Order` is allowed:

`{"guid": 1, "book": {"id": 2, "title": "Tow Sawyer"}}`

Note that type `Order` itself is not open, so it doesn't allow dynamic properties, in contrast to type `Book` .

> **WARNING**
>
> Dynamic properties are not persisted in the underlying data source automatically and must be handled completely by custom code.

Java Type Mapping

Simple Types

The simple values of a deserialized JSON payload can be of type: `String` , `Boolean` , `Number` or simply an `Object` for `null` values.

JSON	Java Type of the `value`
`{"value": "Tom Sawyer"}`	`java.lang.String`
`{"value": true}`	`java.lang.Boolean`
`{"value": 42}`	`java.lang.Number` (Integer)
`{"value": 36.6}`	`java.lang.Number` (BigDecimal)
`{"value": null}`	`java.lang.Object`

Structured Types

The complex and structured types are deserialized to `java.util.Map` , whereas collections are deserialized to `java.util.List` .

JSON	Java Type of the `value`
`{"value": {"name": "Mark Twain"}}`	`java.util.Map<String, Object>`
`{"value":[{"name": "Mark Twain"}, {"name": "Charlotte Bronte"}}]}`	`java.util.List<Map<String, Object>>`

Singletons

A singleton is a special one-element entity introduced in OData V4. You can address it directly by its name from the service root without specifying the entity's keys.

Annotate an entity with `@odata.singleton` or `@odata.singleton.nullable` to use it as a singleton within a service, for example:

```cds
service Sue {
  @odata.singleton entity MySingleton {
    key id : String; // can be omitted in OData v4.01
    prop : String;
    assoc : Association to myEntity;
  }
}
```

You can also define it as an ordered `SELECT` from another entity:

```cds
service Sue {
  @odata.singleton entity OldestEmployee as
    select from Employees order by birthyear;
}
```

Requesting Singletons

As mentioned earlier, you can access singletons without specifying keys in the request URL. They can contain navigation properties, and other entities can include singletons as their navigation properties as well. The `$expand` query option is also supported.

```http
GET …/MySingleton
GET …/MySingleton/prop
GET …/MySingleton/assoc
GET …/MySingleton?$expand=assoc
```

Updating Singletons

The following request updates a *prop* property of a singleton *MySingleton*:

```http
PATCH/PUT …/MySingleton
{prop: “New value”}
```

Deleting Singletons

A `DELETE` request to a singleton is possible only if you annotate a singleton with `@odata.singleton.nullable`. An attempt to delete a singleton annotated with `@odata.singleton` results in an error.

Creating Singletons

Since singletons represent a one-element entity, the system doesn't support a `POST` request.

V2 Support

While CAP defaults to OData V4, the latest protocol version, older projects may need to fall back to OData V2, for example, to keep using existing V2-based UIs.

> **WARNING**
>
> OData V2 is deprecated. Use OData V2 only if you need to support existing UIs or if you need to use specific controls that don't work with V4 *yet*, such as tree tables

(sap.ui.table.TreeTable).

Enabling OData V2 via CDS OData V2 Adapter in Node.js Apps

CAP Node.js supports serving the OData V2 protocol through the *OData V2 adapter for CDS*, which translates between the OData V2 and V4 protocols.

For Node.js projects, add the CDS OData V2 adapter as express.js middleware as follows:

1. Add the adapter package to your project:

   ```sh
   npm add @cap-js-community/odata-v2-adapter
   ```

2. Access OData V2 services at http://localhost:4004/odata/v2/${path} .
3. Access OData V4 services at http://localhost:4004/odata/v4/${path} (as before).

Example: Read service metadata for `CatalogService` :

- CDS:

  ```cds
  @path:'/browse'
  service CatalogService { ... }
  ```

- OData V2: `GET http://localhost:4004/odata/v2/browse/$metadata`
- OData V4: `GET http://localhost:4004/odata/v4/browse/$metadata`

↳ *Find detailed instructions at **@cap-js-community/odata-v2-adapter**.*

Using OData V2 in Java Apps

In CAP Java, serving the OData V2 protocol is supported natively by the CDS OData V2 Adapter.

Miscellaneous

Omitting Elements from APIs

Add annotation *@cds.api.ignore* to suppress unwanted entity fields (for example, foreign key fields) in APIs exposed from the CDS model, that is, OData or OpenAPI. For example:

```cds
entity Books { ...
  @cds.api.ignore
  author : Association to Authors;
}
```

Note that *@cds.api.ignore* is effective on regular elements that are rendered as *Edm.Property* only. The annotation doesn't suppress an *Edm.NavigationProperty*, which is rendered for associations or compositions. If you annotate a managed association, the system propagates the annotations to the (generated) foreign keys. In the previous example, the system mutes the foreign keys of the managed association *author* in the API.

Absolute Context URL

In some scenarios, you need an absolute context URL . In the Node.js runtime, you can achieve this through configuration *cds.odata.contextAbsoluteUrl* .

You can use your own URL (including a protocol and a service path), for example:

```js
cds.odata.contextAbsoluteUrl = "https://your.domain.com/yourService"
```

to customize the annotation as follows:

```json
{
  "@odata.context":"https://your.domain.com/yourService/$metadata#Books(title
  "value":[
    {"ID": 201,"title": "Wuthering Heights","author": "Emily Brontë"},
    {"ID": 207,"title": "Jane Eyre","author": "Charlotte Brontë"},
    {"ID": 251,"title": "The Raven","author": "Edgar Allen Poe"}
  ]
}
```

If you set *`contextAbsoluteUrl`* to something truthy that doesn't match *`http(s)://*`*, the system constructs an absolute path based on the environment of the application on a best effort basis.

We encourage you to stay with the default relative format, if possible, as it's proxy safe.