Conceptual Definition Language (CDL)

The *Conceptual Definition Language (CDL)* is a human-readable language for defining CDS models. Sources are commonly provided in files with `.cds` extensions and get compiled into CSN representations. Following sections provide a reference of all language constructs in CDL, which also serves as a reference of all corresponding CDS concepts and features.

Table of Contents

- Views & Projections
  - The as select from Variant
  - The as projection on Variant
  - Views with Inferred Signatures
  - Virtual elements in views
  - Views with Parameters
- Associations
  - Unmanaged Associations
  - Managed (To-One) Associations
  - To-many Associations
  - Many-to-many Associations
  - Compositions
  - Managed Compositions of Aspects
  - Publish Associations in Projections
- Annotations
  - Annotation Syntax
  - Annotation Targets
  - Annotation Values
  - Records as Syntax Shortcuts
  - Annotation Propagation
  - Expressions as Annotation Values
  - Extend Array Annotations
- Aspects
  - The extend Directive
  - The annotate Directive
  - Named Aspects
  - Includes -- : as Shortcut Syntax
  - Extending Views and Projections
- Services
  - Service Definitions
  - Exposed Entities
  - (Auto-) Redirected Associations
  - Auto-Exposed Entities
  - Custom Actions and Functions
  - Custom-Defined Events
  - Extending Services

Language Preliminaries

Keywords & Identifiers

Keywords in CDL are used to prelude statements, such as imports and namespace directives as well as entity and type declarations. *Identifiers* are used to refer to definitions.

```cds
namespace capire.bookshop;
using { managed, cuid } from '@sap/cds/common';
aspect primary : managed, cuid {}

entity Books : primary {
  title  : String;
  author : Association to Authors;
}

entity Authors : primary {
  name   : String;
}
```

Keywords are *case-insensitive*, but are most commonly used in lowercase notation.

Identifiers are *case-significant*, that is, `Foo` and `foo` would identify different things.

Identifiers have to comply to `/^[$A-Za-z_]\w*$/` or be enclosed in `![ ... ]` like that:

```cds
type ![Delimited Identifier] : String;
```

> **Avoid using delimited identifiers**

> Delimited identifiers in general, but in particular non-ASCII characters, should be avoided as much as possible, for reasons of interoperability.

Built-in Types

The following table lists the built-in types available to all CDS models, and can be used to define entity elements or custom types as follows:

```cds
entity Books {
  key ID : UUID;
  title  : String(111);
  stock  : Integer;
  price  : Price;
}
type Price : Decimal;
```

These types are used to define the structure of entities and services, and are mapped to respective database types when the model is deployed.

CDS Type	Remarks	ANSI SQL [1]
`UUID`	CAP generates RFC 4122 -compliant UUIDs [2]	*NVARCHAR(36)*
`Boolean`	Values: `true`, `false`, `null`, `0`, `1`	*BOOLEAN*
`Integer`	Same as `Int32` by default	*INTEGER*
`Int16`	Signed 16-bit integer, range *[-2^{15} ... $+2^{15}$)*	*SMALLINT*
`Int32`	Signed 32-bit integer, range *[-2^{31} ... $+2^{31}$)*	*INTEGER*
`Int64`	Signed 64-bit integer, range *[-2^{63} ... $+2^{63}$)*	*BIGINT*
`UInt8`	Unsigned 8-bit integer, range *[0 ... 255]*	*TINYINT* [3]
`Decimal ( prec , scale )`	A *decfloat* type is used if arguments are omitted	*DECIMAL*
`Double`	Floating point with binary mantissa	*DOUBLE*
`Date`	e.g. `2022-12-31`	*DATE*
`Time`	e.g. `23:59:59`	*TIME*

CDS Type	Remarks	ANSI SQL [1]
`DateTime`	*sec* precision	*TIMESTAMP*
`Timestamp`	*µs* precision, with up to 7 fractional digits	*TIMESTAMP*
`String ( length )`	Default *length*: 255; on HANA: 5000 [4][5]	*NVARCHAR*
`Binary ( length )`	Default *length*: 255; on HANA: 5000 [4][6]	*VARBINARY*
`LargeBinary`	Unlimited data, usually streamed at runtime	*BLOB*
`LargeString`	Unlimited data, usually streamed at runtime	*NCLOB*
`Map`	Mapped to *NCLOB* for HANA.	*JSON* type
`Vector ( dimension )`	Requires SAP HANA Cloud QRC 1/2024, or later	*REAL_VECTOR*

[1] Concrete mappings to specific databases may differ.
[2] See also Best Practices.
[3] *SMALLINT* on PostgreSQL and H2.
[4] Productive apps should always use an explicit length. Use the default only for rapid prototyping.
[5] Configurable through `cds.cdsc.defaultStringLength`.
[6] Configurable through `cds.cdsc.defaultBinaryLength`.

See also...

↳ *Additional Reuse Types and Aspects by `@sap/cds/common`*
↳ *Mapping to OData EDM types*
↳ *HANA-native Data Types*

Literals

The following literals can be used in CDL (mostly as in JavaScript, Java, and SQL):

```cds
true , false , null         // as in all common languages
11 , 2.4 , 1e3 , 1.23e-11   // for numbers
'A string''s literal'       // for strings
`A string\n paragraph`      // for strings with escape sequences
{ foo:'boo', bar:'car' }    // for records
[ 1, 'two', {three:4} ]     // for arrays
```

↳ *Learn more about literals and their representation in CSN.*

Date & Time Literals

In addition, type-keyword-prefixed strings can be used for date & time literals:

```cds
date'2016-11-24'
time'16:11:32'
timestamp'2016-11-24T12:34:56.789Z'
```

Multiline String Literals

Use string literals enclosed in **single or triple backticks** for multiline strings:

```cds
@escaped: `OK Emoji: \u{1f197}`
@multiline: ```
    This is a CDS multiline string.
    - The indentation is stripped.
    - \u{0055}nicode escape sequences are possible,
      just like common escapes from JavaScript such as
      \r \t \n and more! ```
@data: ```xml
    <main>
      The tag is ignored by the core-compiler but may be
      used for syntax highlighting, similar to markdown.
    </main> ```
entity DocumentedEntity {
  // ...
}
```

> **TIP**
>
> These annotations are illustrative only and are not defined nor have any meaning beyond this example.

Within those strings, escape sequences from JavaScript, such as `\t` or `\u0020`, are supported. Line endings are normalized. If you don't want a line ending at that position, end a line with a backslash (`\`). For string literals inside triple backticks, indentation is stripped and tagging is possible.

Model Imports

The `using` Directive

Using directives allows to import definitions from other CDS models. As shown in line three below you can specify aliases to be used subsequently. You can import single definitions as well as several ones with a common namespace prefix. Optional: Choose a local alias.

using-from.cds

```cds
using foo.bar.scoped.Bar from './contexts';
using foo.bar.scoped.nested from './contexts';
using foo.bar.scoped.nested as specified from './contexts';

entity Car : Bar {}            //> : foo.bar.scoped.Bar
entity Moo : nested.Zoo {}     //> : foo.bar.scoped.nested.Zoo
entity Zoo : specified.Zoo {}  //> : foo.bar.scoped.nested.Zoo
```

Multiple named imports through ES6-like deconstructors:

```cds
using { Foo as Moo, sub.Bar } from './base-model';
entity Boo : Moo { /*...*/ }
entity Car : Bar { /*...*/ }
```

> Also in the deconstructor variant of `using` shown in the previous example, specify fully qualified names.

Model Resolution

Imports in `cds` work very much like `require` in Node.js and `import` s in ES6. In fact, we reuse Node's module loading mechanisms. Hence, the same rules apply:

- Relative path resolution
  Names starting with `./` or `../` are resolved relative to the current model.
- Resolving absolute references
  Names starting with `/` are resolved absolute to the file system.
- Resolving module references
  Names starting with neither `.` nor `/` such as `@sap/cds/common` are fetched for in `node_modules` folders:
  - Files having *.cds*, *.csn*, or *.json* as suffixes, appended in order

- Folders, from either the file set in `cds.main` in the folder's *package.json* or `index.<cds|csn|json>` file.

> **TIP**
>
> To allow for loading from precompiled *.json* files it's recommended to **omit *.cds* suffixes** in import statements, as shown in the provided examples.

Namespaces

The `namespace` Directive

To prefix the names of all subsequent definitions, place a `namespace` directive at the top of a model. This is comparable to other languages, like Java.

namespace.cds

```cds
namespace foo.bar;
entity Foo {}           //> foo.bar.Foo
entity Bar : Foo {}     //> foo.bar.Bar
```

A namespace is not an object of its own. There is no corresponding definition in CSN.

The `context` Directive

Use `contexts` for nested namespace sections.

contexts.cds

```cds
namespace foo.bar;
entity Foo {}           //> foo.bar.Foo
context scoped {
  entity Bar : Foo {}   //> foo.bar.scoped.Bar
  context nested {
    entity Zoo {}       //> foo.bar.scoped.nested.Zoo
  }
}
```

Scoped Definitions

You can define types and entities with other definitions' names as prefixes:

```cds
namespace foo.bar;
entity Foo {}           //> foo.bar.Foo
entity Foo.Bar {}       //> foo.bar.Foo.Bar
type Foo.Bar.Car {}     //> foo.bar.Foo.Bar.Car
```

Fully Qualified Names

A model ultimately is a collection of definitions with unique, fully qualified names. For example, the model in *contexts.cds* would compile to the following CSN:

contexts.json

```json
{"definitions":{
  "foo.bar.Foo": { "kind": "entity" },
  "foo.bar.scoped": { "kind": "context" },
  "foo.bar.scoped.Bar": { "kind": "entity",
    "includes": [ "foo.bar.Foo" ]
  },
  "foo.bar.scoped.nested": { "kind": "context" },
  "foo.bar.scoped.nested.Zoo": { "kind": "entity" }
}}
```

Comments

CDL supports line-end, block comments, and *doc* comments as in Java and JavaScript:

```cds
// line-end comment
/* block comment */
/** doc comment */
```

Doc Comments

A multi-line comment of the form */** … */* at an annotation position is considered a *doc comment*:

```cds
/**
 * I am the description for "Employee"
```

```
 */
entity Employees {
  key ID : Integer;
  /**
   * I am the description for "name"
   */
  name : String;
}
```

The text of a doc comment is stored in CSN in the property `doc`. Doc comments are not propagated. For example, a doc comment defined for an entity isn't automatically copied to projections of this entity. When generating OData EDM(X), doc comments are translated to the annotation `@Core.Description`.

In CAP Node.js, doc comments need to be switched on when calling the compiler:

CLI JavaScript

```sh
cds compile foo.cds --docs
```

> **Doc comments are automatically enabled in CAP Java.**
>
> In CAP Java, doc comments are automatically enabled by the CDS Maven Plugin. In generated interfaces they are converted to corresponding Javadoc comments.

When generating output for deployment to SAP HANA, the first paragraph of a doc comment is translated to the HANA `COMMENT` feature for tables, table columns, and for views (but not for view columns):

```sql
CREATE TABLE Employees (
  ID INTEGER,
  name NVARCHAR(...) COMMENT 'I am the description for "name"'
) COMMENT 'I am the description for "Employee"'
```

Entities & Type Definitions

- Entity Definitions

- Type Definitions
- Structured Types
- Arrayed Types
- Virtual Elements
- Calculated elements
- Default Values
- Type References
- Constraints
- Enums

Entity Definitions

Entities are structured types with named and typed elements, representing sets of (persisted) data that can be read and manipulated using usual CRUD operations. They usually contain one or more designated primary key elements:

```cds
define entity Employees {
  key ID : Integer;
  name : String;
  jobTitle : String;
}
```

> The `define` keyword is optional, that means `define entity Foo` is equal to `entity Foo` .

Type Definitions

You can declare custom types to reuse later on, for example, for elements in entity definitions. Custom-defined types can be simple, that is derived from one of the predefined types, structured types or Associations.

```cds
define type User : String(111);
define type Amount {
  value : Decimal(10,3);
  currency : Currency;
```

```
}
define type Currency : Association to Currencies;
```

> The `define` keyword is optional, that means `define type Foo` is equal to `type Foo`.

↳ *Learn more about **Definitions of Named Aspects**.*

Structured Types

You can declare and use custom struct types as follows:

```cds
type Amount {
  value : Decimal(10,3);
  currency : Currency;
}
entity Books {
  price : Amount;
}
```

Elements can also be specified with anonymous inline struct types. For example, the following is equivalent to the definition of `Books` above:

```cds
entity Books {
  price : {
    value : Decimal(10,3);
    currency : Currency;
  };
}
```

You can declare structured types based on other definitions using the `projection on` syntax. You can use nested projections or aliases as known from entity projections. Only the effective signature of the projection is relevant.

```cds
type CustomerData : projection on Customer {
  name.firstName, // select from structures
  name.lastName,
  address as customerAddress, // aliases
}
```

Arrayed Types

Prefix a type specification with `array of` or `many` to signify array types.

```cds
entity Foo { emails: many String; }
entity Bar { emails: many { kind:String; address:String; }; }
entity Car { emails: many EmailAddress; }
entity Car { emails: EmailAddresses; }
type EmailAddresses : many { kind:String; address:String; }
type EmailAddress : { kind:String; address:String; }
```

> Keywords `many` and `array of` are mere syntax variants with identical semantics and implementations.

When deployed to SQL databases, such fields are mapped to LargeString columns and the data is stored denormalized as JSON array. With OData V4, arrayed types are rendered as `Collection` in the EDM(X).

> **WARNING**
>
> Filter expressions, instance-based authorization and search are not supported on arrayed elements.

Null Values

For arrayed types the `null` and `not null` constraints apply to the *members* of the collections. The default is `not null` indicating that the collections can't hold `null` values.

> **WARNING**
>
> An empty collection is represented by an empty JSON array. A `null` value is invalid for an element with arrayed type.

In the following example the collection `emails` may hold members that are `null`. It may also hold a member where the element `kind` is `null`. The collection `emails` itself must not be `null`!

```cds
entity Bar {
    emails      : many {
        kind    : String null;
```

```
        address : String not null;
    } null;  // -> collection emails may hold null values, overwriting defaul
}
```

Virtual Elements

An element definition can be prefixed with modifier keyword `virtual`. This keyword indicates that this element isn't added to persistent artifacts, that is, tables or views in SQL databases. Virtual elements are part of OData metadata.

By default, virtual elements are annotated with `@Core.Computed: true`, not writable for the client and will be silently ignored. This means also, that they are not accessible in custom event handlers. If you want to make virtual elements writable for the client, you explicitly need to annotate these elements with `@Core.Computed: false`. Still those elements are not persisted and therefore, for example, not sortable or filterable. Further, during read requests, you need to provide values for all virtual elements. You can do this by using post-processing in an `after` handler.

cds
```
entity Employees {
  [...]
  virtual something : String(11);
}
```

Calculated Elements

Elements of entities and aspects can be specified with a calculation expression, in which you can refer to other elements of the same entity/aspect. This can be either a value expression or an expression that resolves to an association.

Calculated elements with a value expression are read-only, no value must be provided for them in a WRITE operation. When reading such a calculated element, the result of the expression is returned. They come in two variants: "on-read" and "on-write". The difference between them is the point in time when the expression is evaluated.

On-read

cds
```
entity Employees {
  firstName : String;
```

```
  lastName : String;
  name : String = firstName || ' ' || lastName;
  name_upper = upper(name);
  addresses : Association to many Addresses;
  city = addresses[kind='home'].city;
}
```

For a calculated element with "on-read" semantics, the calculation expression is evaluated when reading an entry from the entity. Using such a calculated element in a query or view definition is equivalent to writing the expression directly into the query, both with respect to semantics and to performance. In CAP, it is implemented by replacing each occurrence of a calculated element in a query by the respective expression.

Entity using calculated elements:

```cds
entity EmployeeView as select from Employees {
  name,
  city
};
```

Equivalent entity:

```cds
entity EmployeeView as select from Employees {
  firstName || ' ' || lastName as name : String,
  addresses[kind='home'].city as city
};
```

Calculated elements "on-read" are a pure convenience feature. Instead of having to write the same expression several times in queries, you can define a calculated element **once** and then simply refer to it.

In the *definition* of a calculated element "on-read", you can use almost all expressions that are allowed in queries. Some restrictions apply:

- Subqueries are not allowed.
- Nested projections (inline/expand) are not allowed.
- A calculated element can't be key.

Like for views, the expressions are sent unchanged to the database, so you need to ensure that they work on your respective database system(s).

A calculated element can be *used* in every location where an expression can occur. A calculated element can't be used in the following cases:

- in the ON condition of an unmanaged association
- as the foreign key of a managed association
- in a query together with nested projections (inline/expand)

> **WARNING**
>
> For the Node.js runtime, only the new database services under the *@cap-js* scope support this feature.

On-write

Calculated elements "on-write" (also referred to as "stored" calculated elements) are defined by adding the keyword `stored`. A type specification is mandatory.

```cds
entity Employees {
  firstName : String;
  lastName : String;
  name : String = (firstName || ' ' || lastName) stored;
}
```

For a calculated element "on-write", the expression is already evaluated when an entry is written into the database. The resulting value is then stored/persisted like a regular field, and when reading from the entity, it behaves like a regular field as well. Using a stored calculated element can improve performance, in particular when it's used for sorting or filtering. This is paid for by higher memory consumption.

While calculated elements "on-read" are handled entirely by CAP, the "on-write" variant is implemented by using the corresponding feature for database tables. The previous entity definition results in the following table definition:

```sql
-- SAP HANA syntax --
CREATE TABLE Employees (
  firstName NVARCHAR,
  lastName NVARCHAR,
  name NVARCHAR GENERATED ALWAYS AS (firstName || ' ' || lastName)
);
```

For the definition of calculated elements on-write, all the on-read variant's restrictions apply and referencing localized elements isn't allowed. In addition, there are restrictions

that depend on the particular database. Currently all databases supported by CAP have a common restriction: The calculation expression may only refer to fields of the same table row. Therefore, such an expression must not contain subqueries, aggregate functions, or paths with associations.

No restrictions apply for reading a calculated element on-write.

Association-like calculated elements

A calculated element can also define a filtered association/composition using infix filters:

```cds
entity Employees {
  addresses : Association to many Addresses;
  homeAddress = addresses [1: kind='home'];
}
```

For such a calculated element, no explicit type can be specified. Only a single association or composition can occur in the expression, and a filter must be specified.

The effect essentially is like publishing an association with an infix filter.

Default Values

As in SQL you can specify default values to fill in upon INSERTs if no value is specified for a given element.

```cds
entity Foo {
  bar : String default 'bar';
  boo : Integer default 1;
}
```

Default values can also be specified in custom type definitions:

```cds
type CreatedAt : Timestamp default $now;
type Complex {
  real : Decimal default 0.0;
  imag : Decimal default 0.0;
}
```

If the element has an enum type, you can use the enum symbol instead of a literal value:

```cds
type Status : String enum {open; closed;}
entity Order {
  status : Status default #open;
}
```

Type References

If you want to base an element's type on another element of the same structure, you can use the `type of` operator.

```cds
entity Author {
  firstname : String(100);
   lastname : type of firstname; // has type "String(100)"
}
```

For referencing elements of other artifacts, you can use the element access through `:` . Element references with `:` don't require `type of` in front of them.

```cds
entity Employees {
  firstname: Author:firstname;
  lastname: Author:lastname;
}
```

Constraints

Element definitions can be augmented with constraint `not null` as known from SQL.

```cds
entity Employees {
  name : String(111) not null;
}
```

Enums

You can specify enumeration values for a type as a semicolon-delimited list of symbols. For string types, declaration of actual values is optional; if omitted, the actual values are

the string counterparts of the symbols.

```cds
type Gender : String enum { male; female; non_binary = 'non-binary'; }
entity Order {
  status : Integer enum {
    submitted =  1;
    fulfilled =  2;
    shipped   =  3;
    canceled  = -1;
  };
}
```

To enforce your *enum* values during runtime, use the `@assert.range` annotation. For localization of enum values, model them as code list.

Views & Projections

Use `as select from` or `as projection on` to derive new entities from existing ones by projections, very much like views in SQL. When mapped to relational databases, such entities are in fact translated to SQL views but they're frequently also used to declare projections without any SQL views involved.

The entity signature is inferred from the projection.

- The `as select from` Variant
- The `as projection on` Variant
- Views with Inferred Signatures
- Views with Parameters

The `as select from` Variant

Use the `as select from` variant to use all possible features an underlying relational database would support using any valid CQL query including all query clauses.

```cds
entity Foo1 as select from Bar; //> implicit {*}
entity Foo2 as select from Employees { * };
entity Foo3 as select from Employees LEFT JOIN Bar on Employees.ID=Bar.ID {
  foo, bar as car, sum(boo) as moo
} where exists (
  SELECT 1 as anyXY from SomeOtherEntity as soe where soe.x = y
)
group by foo, bar
order by moo asc;
```

The *as projection on* Variant

Use the *as projection on* variant instead of *as select from* to indicate that you don't use the full power of SQL in your query. For example, having a restricted query in an entity allows us to serve such an entity from external OData services.

```cds
entity Foo as projection on Bar {...}
```

Currently, the restrictions of *as projection on* compared to *as select from* are:

- no explicit, manual *JOINs*
- no explicit, manual *UNIONs*
- no sub selects in from clauses

Over time, we can add additional checks depending on specific outbound protocols.

Views with Inferred Signatures

By default views inherit all properties and annotations from their primary underlying base entity. Their *elements* signature is **inferred** from the projection on base elements. Each element inherits all properties from the respective base element, except the *key* property. The *key* property is only inherited if all of the following applies:

- No explicit *key* is set in the query.
- All key elements of the primary base entity are selected (for example, by using ***).
- No path expression with a to-many association is used.

- No *union* , *join* or similar query construct is used.

For example, the following definition:

```cds
entity SomeView as select from Employees {
  ID,
  name,
  job.title as jobTitle
};
```

Might result in this inferred signature:

```cds
entity SomeView {
  key ID: Integer;
  name: String;
  jobTitle: String;
};
```

Note: CAP does **not** enforce uniqueness for key elements of a view or projection.

Use a CDL cast to set an element's type, if one of the following conditions apply:

- You don't want to use the inferred type.
- The query column is an expression (no inferred type is computed).

```cds
entity SomeView as select from Employees {
  ID : Integer64,
  name : LargeString,
  'SAP SE' as company : String
};
```

> **TIP**
>
> By using a cast, annotations and other properties are inherited from the provided type and not the base element, see Annotation Propagation

Virtual elements in views

Virtual elements can be defined in views or projections like this:

```cds
entity SomeView as select from Employee {
  // ...,
  virtual virt1 : String(22),
  virtual virt2  // virtual element without type
}
```

These virtual elements have no relation to the query source `Employee` but are new fields in the view. Virtual elements in views or projections are handled as described in the section on virtual elements in entities.

Views with Parameters

You can equip views with parameters that are passed in whenever that view is queried. Default values can be specified. Refer to these parameters in the view's query using the prefix `:` .

```cds
entity SomeView ( foo: Integer, bar: Boolean )
as SELECT * from Employees where ID=:foo;
```

When selecting from a view with parameters, the parameters are passed by name. In the following example, `UsingView` also has a parameter `bar` that is passed down to `SomeView` .

```cds
entity UsingView ( bar: Boolean )
as SELECT * from SomeView(foo: 17, bar: :bar);
```

For Node.js, there's no programmatic API yet. You need to provide a CQN snippet.

In CAP Java, run a select statement against the view with named parameter values:

Node Java

```js
SELECT.from({ ref: [{ id: 'UsingView', args: { bar: { val: true }}} ]} )
```

↳ *Learn more about how to expose views with parameters in **Services - Exposed Entities**.*

↳ *Learn more about views with parameters for existing HANA artifacts in **Native SAP HANA Artifacts**.*

Associations

Associations capture relationships between entities. They are like forward-declared joins added to a table definition in SQL.

- Unmanaged Associations
- Managed Associations
- To-many Associations
- Many-to-many Associations
- Compositions
- Managed Compositions

Unmanaged Associations

Unmanaged associations specify arbitrary join conditions in their `on` clause, which refer to available foreign key elements. The association's name (`address` in the following example) is used as the alias for the to-be-joined target entity.

```cds
entity Employees {
  address : Association to Addresses on address.ID = address_ID;
  address_ID : Integer;  //> foreign key
}
```

```cds
entity Addresses {
  key ID : Integer;
}
```

Managed (To-One) Associations

For to-one associations, CDS can automatically resolve and add requisite foreign key elements from the target's primary keys and implicitly add respective join conditions.

```cds
entity Employees {
  address : Association to Addresses;
```

```
}
```

This example is equivalent to the unmanaged example above, with the foreign key element `address_ID` being added automatically upon activation to a SQL database. The names of the automatically added foreign key elements cannot be changed.

> Note: For adding foreign key constraints on database level, see Database Constraints..

If the target has a single primary key, a default value can be provided. This default applies to the generated foreign key element `address_ID` :

cds
```
entity Employees {
  address : Association to Addresses default 17;
}
```

To-many Associations

For to-many associations specify an `on` condition following the canonical expression pattern `<assoc>.<backlink> = $self` as in this example:

cds
```
entity Employees {
  key ID : Integer;
  addresses : Association to many Addresses
    on addresses.owner = $self;
}
```

cds
```
entity Addresses {
  owner : Association to Employees;  //> the backlink
}
```

> The backlink can be any managed to-one association on the *many* side pointing back to the *one* side.

Many-to-many Associations

For many-to-many association, follow the common practice of resolving logical many-to-many relationships into two one-to-many associations using a link entity to connect both. For example:

```cds
entity Employees { [...]
  addresses : Association to many Emp2Addr on addresses.emp = $self;
}
entity Emp2Addr {
  key emp : Association to Employees;
  key adr : Association to Addresses;
}
```

↳ *Learn more about **Managed Compositions for Many-to-many Relationships**.*

Compositions

Compositions constitute document structures through *contained-in* relationships. They frequently show up in to-many header-child scenarios.

```cds
entity Orders {
  key ID: Integer; //...
  Items : Composition of many Orders.Items on Items.parent = $self;
}
entity Orders.Items {
  key pos : Integer;
  key parent : Association to Orders;
  product : Association to Products;
  quantity : Integer;
}
```

Contained-in relationship

Essentially, Compositions are the same as *associations*, just with the additional information that this association represents a *contained-in* relationship so the same syntax and rules apply in their base form.

> **Limitations of Compositions of one**
>
> Using compositions of one for entities is discouraged. There is often no added value of using them as the information can be placed in the root entity. Compositions of one have limitations as follow:
>
> - Very limited Draft support. Fiori elements does not support compositions of one unless you take care of their creation in a custom handler.

- No extensive support for modifications over paths if compositions of one are involved. You must fill in foreign keys manually in a custom handler.

Managed Compositions of Aspects

Use managed compositions variant to nicely reflect document structures in your domain models, without the need for separate entities, reverse associations, and unmanaged *on* conditions.

With Inline Targets

```cds
entity Orders {
  key ID: Integer; //...
  Items : Composition of many {
    key pos : Integer;
    product : Association to Products;
    quantity : Integer;
  }
};
```

Managed Compositions are mostly syntactical sugar: Behind the scenes, they are unfolded to the unmanaged equivalent as shown above by automatically adding a new entity, the name of which being constructed as a scoped name from the name of parent entity, followed by the name of the composition element, that is *Orders.Items* in the previous example. You can safely use this name at other places, for example to define an association to the generated child entity:

```cds
entity Orders {
  // …
  specialItem : Association to Orders.Items;
};
```

With Named Targets

Instead of anonymous target aspects you can also specify named aspects, which are unfolded the same way as anonymous inner types, as shown in the previous example:

```cds
entity Orders {
  key ID: Integer; //...
```

```
  Items : Composition of many OrderItems;
}
aspect OrderItems {
  key pos : Integer;
  product : Association to Products;
  quantity : Integer;
}
```

Default Target Cardinality

If not otherwise specified, a managed composition of an aspect has the default target cardinality *to-one*.

For Many-to-many Relationships

Managed Compositions are handy for many-to-many relationships, where a link table usually is private to one side.

```cds
entity Teams { [...]
  members : Composition of many { key user: Association to Users; }
}
entity Users { [...]
  teams: Association to many Teams.members on teams.user = $self;
}
```

And here's an example of an attributed many-to-many relationship:

```cds
entity Teams { [...]
  members : Composition of many {
    key user : Association to Users;
    role : String enum { Lead; Member; Collaborator; }
  }
}
entity Users { ... }
```

To navigate between *Teams* and *Users*, you have to follow two associations: `members.user` or `teams.up_`. In OData, to get all users of all teams, use a query like the following:

```cds
GET /Teams?$expand=members($expand=user)
```

Publish Associations in Projections

As associations are first class citizens, you can put them into the select list of a view or projection ("publish") like regular elements. A `select *` includes all associations. If you need to rename an association, you can provide an alias.

```cds
entity P_Employees as projection on Employees {
  ID,
  addresses
}
```

The effective signature of the projection contains an association `addresses` with the same properties as association `addresses` of entity `Employees`.

Publish Associations with Infix Filter

When publishing an unmanaged association in a view or projection, you can add a filter condition. The ON condition of the resulting association is the ON condition of the original association plus the filter condition, combined with `and`.

```cds
entity P_Authors as projection on Authors {
  *,
  books[stock > 0] as availableBooks
};
```

In this example, in addition to `books` projection `P_Authors` has a new association `availableBooks` that points only to those books where `stock > 0`.

If the filter condition effectively reduces the cardinality of the association to one, you should make this explicit in the filter by adding a `1:` before the condition:

```cds
entity P_Employees as projection on Employees {
  *,
  addresses[1: kind='home'] as homeAddress  // homeAddress is to-one
}
```

Filters usually are provided only for to-many associations, which usually are unmanaged. Thus publishing with a filter is almost exclusively used for unmanaged associations. Nevertheless you can also publish a managed association with a filter. This will automatically turn the resulting association into an unmanaged one. You must ensure that all foreign key elements needed for the ON condition are explicitly published.

```cds
entity P_Books as projection on Books {
  author.ID as authorID,  // needed for ON condition of deadAuthor
  author[dateOfDeath is not null] as deadAuthor  // -> unmanaged association
};
```

Publishing a *composition* with a filter is similar, with an important difference: in a deep Update, Insert, or Delete statement the respective operation does not cascade to the target entities. Thus the type of the resulting element is set to `cds.Association`.

↳ *Learn more about `cds.Association`.*

In SAP Fiori Draft, it behaves like an "enclosed" association, that means, it points to the target draft entity.

In the following example, `singleItem` has type `cds.Association`. In draft mode, navigating along `singleItems` doesn't leave the draft tree.

```cds
@odata.draft.enabled
entity P_orders as projection on Orders {
  *,
  Items[quantity = 1] as singleItems
}
```

Annotations

This section describes how to add Annotations to model definitions written in CDL, focused on the common syntax options, and fundamental concepts. Find additional information in the OData Annotations guide.

- Annotation Syntax
- Annotation Targets
- Annotation Values
- Expressions as Annotation Values
- Records as Syntax Shortcuts
- Annotation Propagation

Annotation Syntax

Annotations in CDL are prefixed with an `@` character and can be placed before a definition, after the defined name or at the end of simple definitions.

```cds
@before entity Foo @inner {
  @before simpleElement @inner : String @after;
  @before structElement @inner { /* elements */ }
}
```

Multiple annotations can be placed in each spot separated by whitespaces or enclosed in `@(...)` and separated by comma - like the following are equivalent:

```cds
entity Foo @(
  my.annotation: foo,
  another.one: 4711
) { /* elements */ }
```

```cds
@my.annotation:foo
@another.one: 4711
entity Foo { /* elements */ }
```

For an `@inner` annotation, only the syntax `@(...)` is available.

Using `annotate` Directives

Instead of interspersing annotations with definitions, you can also use the `annotate` directive to add annotations to existing definitions.

```cds
annotate Foo with @(
  my.annotation: foo,
  another.one: 4711
);
```

↳ *Learn more about the `annotate` directive in the **Aspects** chapter below.*

Annotation Targets

You can basically annotate any named thing in a CDS model, such as:

Contexts and services:

```cds
@before context foo.bar @inner { ... }
@before service Sue @inner { ... }
```

Definitions and elements with simple or struct types:

```cds
@before type Foo @inner : String @after;
@before entity Foo @inner {
  @before key ID @inner : String @after;
  @before title @inner : String @after;
  @before struct @inner { ...elements... };
}
```

Enums:

```cds
… status : String @inner enum {
  open @after;
  closed @after;
  cancelled @after;
  accepted @after;
  rejected @after;
}
```

Columns in a view definition's query:

```cds
… as select from Foo {
  @before expr as alias @inner : String,
  …
}
```

Parameters in view definitions:

```cds
… with parameters (
  @before param @(inner) : String @after
) …
```

Actions/functions including their parameters and result:

```cds
@before action doSomething @inner (
  @before param @(inner) : String @after
) returns @before resultType;
```

Or in case of a structured result:

```cds
action doSomething() returns @before {
  @before resultElem @inner : String @after;
};
```

Annotation Values

Values can be literals, references, or expressions. Expressions are explained in more detail in the next section. If no value is given, the default value is `true` as for `@aFlag` in the following example:

```cds
@aFlag //= true, if no value is given
@aBoolean: false
@aString: 'foo'
@anInteger: 11
@aDecimal: 11.1
@aSymbol: #foo
@aReference: foo.bar
@anArray: [ /* can contain any kind of value */ ]
@anExpression: ( foo.bar * 17 )  // expression, see next section
```

As described in the CSN spec, the previously mentioned annotations would compile to CSN as follows:

```jsonc
{
  "@aFlag": true,
  "@aBoolean": false,
  "@aString": "foo",
  "@anInteger": 11,
  "@aDecimal": 11.1,
  "@aSymbol": {"#":"foo"},
  "@aReference": {"=":"foo.bar"},
  "@anArray": [ /* … */ ],
```

```
  "@anExpression": { /* see next section */ }
}
```

> **TIP**
>
> In contrast to references in expressions, plain references aren't checked, resolved, or rewritten by CDS parsers or linkers. They're interpreted and evaluated only on consumption-specific modules. For example, for SAP Fiori models, it's the *4odata* and *2edm(x)* processors.

Records as Syntax Shortcuts

Annotations in CDS are flat lists of key-value pairs assigned to a target. The record syntax - that is, *`{key:<value>, ...}`* - is a shortcut notation that applies a common prefix to nested annotations. For example, the following are equivalent:

```cds
@Common.foo.bar
@Common.foo.car: 'wheels'
```

```cds
@Common: { foo.bar, foo.car: 'wheels' }
```

```cds
@Common.foo: { bar }
@Common.foo.car: 'wheels'
```

```cds
@Common.foo: { bar, car: 'wheels' }
```

and they would show up as follows in a parsed model (→ see CSN):

```json
{
  "@Common.foo.bar": true,
  "@Common.foo.car": "wheels"
}
```

Annotation Propagation

Annotations are inherited from types and base types to derived types, entities, and elements as well as from elements of underlying entities in case of views.

For example, given this view definition:

```cds
using Books from './bookshop-model';
entity BooksList as select from Books {
  ID, genre : Genre, title,
  author.name as author
};
```

- `BooksList` would inherit annotations from `Books`
- `BooksList:ID` would inherit from `Books:ID`
- `BooksList:author` would inherit from `Books:author.name`
- `BooksList.genre` would inherit from type `Genre`

The rules are:

1. Entity-level properties and annotations are inherited from the **primary** underlying source entity — here `Books` .
2. Each element that can **unambiguously** be traced back to a single source element, inherits that element's properties.
3. An explicit **cast** in the select clause cuts off the inheritance, for example, as for `genre` in our previous example.

> **TIP**
>
> Propagation of annotations can be stopped via value `null` , for example, `@anno: null` .

Expressions as Annotation Values

In order to use an expression as an annotation value, it must be enclosed in parentheses:

```cds
@anExpression: ( foo.bar * 11 )
```

Syntactically, the same expressions are supported as in a select item or in the where clause of a query, except subqueries. The expression can of course also be a single reference or a simple value:

```cds
@aRefExpr: ( foo.bar )
@aValueExpr: ( 11 )
```

Some advantages of using expressions as "first class" annotation values are:

- syntax and references are checked by the compiler
- code completion
- automatic path rewriting in propagated annotations
- automatic translation of expressions in OData annotations

Limitations

Elements that are not available to the compiler, for example the OData draft decoration, can't be used in annotation expressions.

Name resolution

Each path in the expression is checked:

- For an annotation assigned to an entity, the first path step is resolved as element of the entity.
- For an annotation assigned to an entity element, the first path step is resolved as the annotated element or its siblings.
- If the annotation is assigned to a subelement of a structured element, the top level elements of the entity can be accessed via `$self` .
- A parameter `par` can be accessed via `:par` , just like parameters of a parametrized entity in queries.
- For an annotation assigned to a bound action or function, elements of the respective entity can be accessed via `$self` .
- The draft-specific elements `IsActiveEntity` , `HasActiveEntity` , and `HasDraftEntity` can be referred to with respective magic variables `$draft.IsActiveEntity` , `$draft.HasActiveEntity` , and `$draft.HasDraftEntity` . During draft augmentation, `$draft.<...>` is rewritten to `$self.<...>` for all draft enabled entities (root and sub nodes, but not for named types or entity parameters).
- If a path can't be resolved successfully, compilation fails with an error.

In contrast to `@aReference: foo.bar` , a single reference written as expression `@aRefExpr: ( foo.bar )` is checked by the compiler.

```cds
@MyAnno: (a)            // reference to element
entity Foo (par: Integer) {
  key ID : Integer;
  @MyAnno: (:par)       // reference to entity parameter
  a : Integer;
  @MyAnno: (a)          // reference to sibling element
  b : Integer;
  s {
    @MyAnno: (y)        // reference to sibling element
    x : Integer;
    @MyAnno: ($self.a)  // reference to top level element
    y : Integer;
  }
}
actions {
  @MyAnno: ($self.a)
  action A ()
}
```

CSN Representation

In CSN, the expression is represented as a record with two properties:

- A string representation of the expression is stored in property `=`.
- A tokenized representation of the expression is stored in one of the properties *`xpr`*, *`ref`*, *`val`*, *`func`*, etc. (like if the expression was written in a query).

```json
{
  "@anExpression": {
    "=": "foo.bar * 11",
    "xpr": [ {"ref": ["foo", "bar"]}, "*", {"value": 11} ]
  },
  "@aRefExpr": {
    "=": "foo.bar",
    "ref": ["foo", "bar"]
  },
  "@aValueExpr": {
    "=": "11",
    "val": 11
  }
}
```

Note the different CSN representations for a plain value `"@anInteger": 11` and a value written as expression `@aValueExpr: ( 11 )`, respectively.

Propagation

Annotations are propagated in views/projections, via includes, and along type references. If the annotation value is an expression, it is sometimes necessary to adapt references inside the expression during propagation, for example, when a referenced element is renamed in a projection. The compiler automatically takes care of the necessary rewriting. When a reference in an annotation expression is rewritten, the `=` property is adapted accordingly if the expression is a single reference, otherwise it is set to `true`.

Example:

```cds
entity E {
  @Common.Text: (text)
  code : Integer;
  text : String;
}

entity P as projection on E {
  code,
  text as descr
}
```

When propagated to element `code` of projection `P`, the annotation is automatically rewritten to `@Common.Text: (descr)`.

▶ *Resulting CSN*

> **INFO**
>
> There may be situations where automatic rewriting doesn't work, resulting in the compiler error `anno-missing-rewrite`. In these cases you can overwrite the annotation with the correct expression in the new location.

CDS Annotations

Using an expression as annotation value only makes sense if the evaluator of the annotation is prepared to deal with the new CSN representation. Currently, the CAP

runtimes only support expressions in the *`where`* property of the *`@restrict`* annotation.

```cds
entity Orders @(restrict: [
    { grant: 'READ', to: 'Auditor', where: (AuditBy = $user.id) }
  ]) {/*...*/}
```

More annotations are going to follow in upcoming releases.

Of course, you can use this feature also in your custom annotations, where you control the code that evaluates the annotations.

OData Annotations

The OData backend of the CAP CDS compiler supports expression-valued annotations. See Expressions in OData Annotations.

Extend Array Annotations

Usually, the annotation value provided in an *`annotate`* directive overwrites an already existing annotation value.

If the existing value is an array, the *ellipsis* syntax allows to insert new values **before** or **after** the existing entries, instead of overwriting the complete array. The ellipsis represents the already existing array entries. Of course, this works with any kind of array entries.

This is a sample of an existing array:

```cds
@anArray: [3, 4] entity Foo { /* elements */ }
```

This shows how to extend the array:

```cds
annotate Foo with @anArray: [1, 2, ...];  //> prepend new values: [1, 2, 3, 4]
annotate Foo with @anArray: [..., 5, 6];  //> append new values: [3, 4, 5, 6]
annotate Foo with @anArray: [1, 2, ..., 5, 6]; //> prepend and append
```

It's also possible to insert new entries at **arbitrary positions**. For this, use *`... up to`* with a *comparator* value that identifies the insertion point.

```cds
[... up to <comparator>, newEntry, ...]
```

`... up to` represents the existing entries of the array from the current position up to and including the first entry that matches the comparator. New entries are then inserted behind the matched entry. If there's no match, new entries are appended at the end of the existing array.

This is a sample of an existing array:

```cds
@anArray: [1, 2, 3, 4, 5, 6] entity Bar { /* elements */ }
```

This shows how to insert values after `2` and `4`:

```cds
annotate Bar with @anArray: [
  ... up to 2,  // existing entries 1, 2
   2.1, 2.2,    // insert new entries 2.1, 2.2
  ... up to 4,  // existing entries 3, 4
  4.1, 4.2,     // insert new entries 4.1, 4.2
  ...           // remaining existing entries 5, 6
];
```

The resulting array is:

```js
[1, 2, 2.1, 2.2, 3, 4, 4.1, 4.2, 5, 6]
```

If your array entries are objects, you have to provide a *comparator object*. It matches an existing entry, if all attributes provided in the comparator match the corresponding attributes in an existing entry. The comparator object doesn't have to contain all attributes that the existing array entries have, simply choose those attributes that sufficiently characterize the array entry after which you want to insert. Only simple values are allowed for the comparator attributes.

Example: Insert a new entry after `BeginDate`.

```cds
@UI.LineItem: [
    { $Type: 'UI.DataFieldForAction', Action: 'TravelService.acceptTravel', La
    { Value: TravelID,  Label: 'ID'    },
    { Value: BeginDate, Label: 'Begin' },
    { Value: EndDate,   Label: 'End'   }
  ]
entity TravelService.Travel { /* elements */ }
```

For this, you provide a comparator object with the attribute `Value`:

```cds
annotate TravelService.Travel with @UI.LineItem: [
  ... up to { Value: BeginDate },  // ... up to with comparator object
  { Value: BeginWeekday, Label: 'Day of week' }, // new entry
  ... // remaining array entries
];
```

> **TIP**
>
> Only direct annotations can be extended using `...`. It's not supported to extend propagated annotations, for example, from aspects or types.

Aspects

CDS's aspects allow to flexibly extend definitions by new elements as well as overriding properties and annotations. They're based on a mixin approach as known from Aspect-oriented Programming methods.

- The *extend* Directive
- The *annotate* Directive
- Named Aspects
- Shortcut Syntax *:*
- Extending Views / Projections
- See also: Aspect-oriented Modelling

The *extend* Directive

Use *extend* to add extension fields or to add/override metadata to existing definitions, for example, annotations, as follows:

```cds
extend Foo with @title:'Foo';
extend Bar with @title:'Bar' {
```

```
  newField : String;
  extend nestedStructField {
    newField : String;
    extend existingField @title:'Nested Field';
  }
}
```

▶ *Note the nested `extend` for existing fields*

You can also directly extend a single element:

```cds
extend Foo:nestedStructField with { newField : String; }
```

With `extend` you can enlarge the *length* of a String or *precision* and *scale* of a Decimal:

```cds
extend User with (length:120);
extend Books:price.value with (precision:12,scale:3);
```

The extended type or element directly must have the respective property.

For multiple conflicting `extend` statements, the last `extend` wins, that means in three files `a.cds <- b.cds <- c.cds`, where `<-` means `using from`, the `extend` from `c.cds` is applied, as it is the last in the dependency chain.

The `annotate` Directive

The `annotate` directive allows to annotate already existing definitions that may have been imported from other files or projects.

```cds
annotate Foo with @title:'Foo';
annotate Bar with @title:'Bar' {
  nestedStructField {
    existingField @title:'Nested Field';
  }
}
```

▶ *`annotate` is a shortcut for `extend` ...*

You can also directly annotate a single element:

```cds
annotate Foo:existingField @title: 'Simple Field';
annotate Foo:nestedStructField.existingField @title:'Nested Field';
```

Named Aspects

You can use *extend* with predefined aspects, to apply the same extensions to multiple targets:

```cds
@annotation
aspect NamedAspect {
  created { at: Timestamp; _by: User; }
} actions {
  action A() returns String;
}
```

```cds
extend Foo with NamedAspect;
extend Bar with NamedAspect;
```

By extending an entity with an aspect, you add all the aspect's fields, actions, and annotations to the entity.

Use keyword *aspect* as shown in the example to declare definitions that are only meant to be used in such extensions, not as types for elements.

To reuse annotations, without adding elements, use an empty aspect and extend your target with it You can even extend projections with such aspects.

```cds
@annotation
aspect ReuseAnnotations {};
entity Proj as projection on Bar;
```

```cds
extend Proj with ReuseAnnotations;
```

Includes -- *:* as Shortcut Syntax

You can use an inheritance-like syntax option to extend a definition with one or more named aspects as follows:

```cds
define entity Foo : SomeAspect, AnotherAspect {
  key ID : Integer;
  name : String;
  [...]
}
```

This is syntactical sugar and equivalent to using a sequence of extends as follows:

```cds
define entity Foo {}
extend Foo with SomeAspect;
extend Foo with AnotherAspect;
extend Foo with {
  key ID : Integer;
  name : String;
  [...]
}
```

You can apply this to any definition of an entity or a structured type.

Extending Views and Projections

Use the *extend <entity> with columns* variant to extend the select list of a projection or view entity and do the following:

- Include more elements existing in the underlying entity.
- Add new calculated fields.
- Add new unmanaged associations.

```cds
extend SomeView with columns {
  foo as moo @woo,
  1 + 1 as two,
  bar : Association to Bar on bar.ID = moo
}
```

Enhancing nested structs isn't supported. Furthermore, the table alias of the view's data source is not accessible in such an extend.

You can use the common *annotate* directive to just add/override annotations of a view's elements.

Services

Service Definitions

CDS allows to define service interfaces as collections of exposed entities enclosed in a `service` block, which essentially is and acts the same as `context`:

```cds
service SomeService {
  entity SomeExposedEntity { ... };
  entity AnotherExposedEntity { ... };
}
```

The endpoint of the exposed service is constructed by its name, following some conventions (the string `service` is dropped and kebab-case is enforced). If you want to overwrite the path, you can add the `@path` annotation as follows:

```cds
@path: 'myCustomServicePath'
service SomeService { ... }
```

↳ *Watch a short video by DJ Adams on how the `@path` annotations works.*

Exposed Entities

The entities exposed by a service are most frequently projections on entities from underlying data models. Standard view definitions, using `as select from` or `as projection on`, can be used for exposing entities.

```cds
service CatalogService {
  entity Product as projection on data.Products {
    *, created.at as since
  } excluding { created };
}
service MyOrders {
  //> $user only implemented for SAP HANA
  entity Order as select from data.Orders { * } where buyer=$user.id;
  entity Product as projection on CatalogService.Product;
}
```

> **TIP**
>
> You can optionally add annotations such as `@readonly` or `@insertonly` to exposed entities, which will be enforced by the CAP runtimes in Java and Node.js.

Entities can be also exposed as views with parameters:

```cds
service MyOrders {
  entity OrderWithParameter( foo: Integer ) as select from data.Orders where
}
```

A parametrized view like modeled in the section on `view with parameter` can be exposed as follows:

```cds
service SomeService {
  entity ViewInService( p1: Integer, p2: Boolean ) as select from data.SomeVi
}
```

Then the OData request for views with parameters should look like this:

```cds
GET: /OrderWithParameter(foo=5)/Set or GET: /OrderWithParameter(5)/Set
GET: /ViewInService(p1=5, p2=true)/Set
```

To expose an entity, it's not necessary to be lexically enclosed in the service definition. An entity's affiliation to a service is established using its fully qualified name, so you can

also use one of the following options:

- Add a namespace.
- Use the service name as prefix.

In the following example, all entities belong to/are exposed by the same service:

myservice.cds

```cds
service foo.MyService {
  entity A { /*...*/ };
}
entity foo.MyService.B { /*...*/ };
```

another.cds

```cds
namespace foo.MyService;
entity C { /*...*/ };
```

(Auto-) Redirected Associations

When exposing related entities, associations are automatically redirected. This ensures that clients can navigate between projected entities as expected. For example:

```cds
service AdminService {
  entity Books as projection on my.Books;
  entity Authors as projection on my.Authors;
  //> AdminService.Authors.books refers to AdminService.Books
}
```

Resolving Ambiguities

Auto-redirection fails if a target can't be resolved unambiguously, that is, when there is more than one projection with the same minimal 'distance' to the source. For example, compiling the following model with two projections on `my.Books` would produce this error:

> **DANGER**

> Target "Books" is exposed in service "AdminService" by multiple projections "AdminService.ListOfBooks", "AdminService.Books" - no implicit redirection.

```cds
service AdminService {
  entity ListOfBooks as projection on my.Books;
  entity Books as projection on my.Books;
  entity Authors as projection on my.Authors;
  //> which one should AdminService.Authors.books refer to?
}
```

Using *redirected to* with Projected Associations

You can use *redirected to* to resolve the ambiguity as follows:

```cds
service AdminService {
  entity ListOfBooks as projection on my.Books;
  entity Books as projection on my.Books;
  entity Authors as projection on my.Authors { *,
    books : redirected to Books //> resolved ambiguity
  };
}
```

Using *@cds.redirection.target* Annotations

Alternatively, you can use the boolean annotation *@cds.redirection.target* with value *true* to make an entity a preferred redirection target, or with value *false* to exclude an entity as target for auto-redirection.

```cds
service AdminService {
  @cds.redirection.target: true
  entity ListOfBooks as projection on my.Books;
  entity Books as projection on my.Books;
  entity Authors as projection on my.Authors;
}
```

Auto-Exposed Entities

Annotate entities with `@cds.autoexpose` to automatically expose them in services containing entities with associations referring to them.

For example, given the following entity definitions:

```cds
// schema.cds
namespace schema;
entity Bar @cds.autoexpose { key id: Integer; }

using { sap.common.CodeList } from '@sap/cds/common';
entity Car : CodeList { key code: Integer; }
//> inherits  @cds.autoexpose from  sap.common.CodeList
```

... a service definition like this:

```cds
using { schema as my } from './schema.cds';
service Zoo {
  entity Foo { //...
    bar : Association to my.Bar;
    car : Association to my.Car;
  }
}
```

... would result in the service being automatically extended like this:

```cds
extend service Zoo with { // auto-exposed entities:
   @readonly entity Foo_bar as projection on Bar;
   @readonly entity Foo_car as projection on Car;
}
```

You can still expose such entities explicitly, for example, to make them read-write:

```cds
service Sue {
  entity Foo { /*...*/ }
  entity Bar as projection on my.Bar;
}
```

↳ *Learn more about **CodeLists in** `@sap/cds/common` .*

Custom Actions and Functions

Within service definitions, you can additionally specify *actions* and *functions*. Use a comma-separated list of named and typed inbound parameters (optional) and a response type (optional for actions), which can be either a:

- Predefined Type
- Reference to a custom-defined type
- Inline definition of an anonymous structured type

```cds
service MyOrders {
  entity Order { /*...*/ };
  // unbound actions / functions
  type cancelOrderRet {
    acknowledge: String enum { succeeded; failed; };
    message: String;
  }
  action cancelOrder ( orderID:Integer, reason:String ) returns cancelOrderRe
  function countOrders() returns Integer;
  function getOpenOrders() returns array of Order;
}
```

> **TIP**
>
> The notion of actions and functions in CDS adopts that of OData ; actions and functions on service-level are *unbound* ones.

Bound Actions and Functions

Actions and functions can also be bound to individual entities of a service, enclosed in an additional *actions* block as the last clause in an entity/view definition.

```cds
service CatalogService {
  entity Products as projection on data.Products { ... }
    actions {
      // bound actions/functions
      action addRating (stars: Integer);
      function getViewsCount() returns Integer;
    }
}
```

Bound actions and functions have a binding parameter that is usually implicit. It can also be modeled explicitly: the first parameter of a bound action or function is treated as binding parameter, if it's typed with `$self` or `many $self`. Use the keyword `many` to indicate that the action or function is bound to a collection of instances rather than to a single one. Also use the binding parameter to control its name.

```cds
service CatalogService {
  entity Products as projection on data.Products { ... }
    actions {
      // bound actions/functions with explicit binding parameter
      action A1 (prod: $self, stars: Integer);
      action A2 (in: many $self);  // bound to collection of Products
    }
}
```

Explicitly modelled binding parameters are ignored for OData V2.

Returning Media Data Streams

Actions and functions can also be modeled to return streamed media data such as images and CSV files. To achieve this, the return type of the actions or functions must refer to a predefined type, annotated with media data annotations, that is defined in the same service. The minimum set of annotations required is `@Core.MediaType`.

```cds
service CatalogService {
  @Core.MediaType: 'image/png'  @Core.ContentDisposition.Filename: 'image.png
  type png : LargeBinary;

  entity Products as projection on data.Products { ... }
    actions {
      function image() returns png;
    }
}
```

Custom-Defined Events

Similar to Actions and Functions you can declare `events`, which a service emits via messaging channels. Essentially, an event declaration looks very much like a type

definition, specifying the event's name and the type structure of the event messages' payload.

```cds
service MyOrders { ...
  event OrderCanceled {
    orderID: Integer;
    reason: String;
  }
}
```

An event can also be defined as projection on an entity, structured type, or another event. Only the effective signature of the projection is relevant.

```cds
service MyOrders { ...
  event OrderCanceledNarrow : projection on OrderCanceled { orderID }
}
```

Extending Services

You can extend services with additional entities and actions much as you would add new entities to a context:

```cds
extend service CatalogService with {
  entity Foo {};
  function getRatings() returns Integer;
}
```

Similarly, you can extend entities with additional actions as you would add new elements:

```cds
extend entity CatalogService.Products with actions {
  function getRatings() returns Integer;
}
```

CDS

Schema Notation (CSN)

Was this page helpful?