Core Schema Notation (CSN)

CSN (pronounced as "*Season*") is a notation for compact representations of CDS models — tailored to serve as an optimized format to share and interpret models with minimal footprint and dependencies.

It's similar to JSON Schema but goes beyond JSON's abilities, in order to capture full-blown *Entity-Relationship Models* and Extensions. This makes CSN models a perfect source to generate target models, such as OData/EDM or OpenAPI interfaces, as well as persistence models for SQL or NoSQL databases.

Table of Contents

Anatomy

A CSN model in **JSON**:

```json
{
  "requires": [ "@sap/cds/common", "./db/schema" ],
  "definitions": {
    "some.type": { "type": "cds.String", "length": 11 },
    "another.type": { "type": "some.type" },
    "structured.type": { "elements": {
      "foo": { "type": "cds.Integer" },
      "bar": { "type": "cds.String" }
    }}
  },
  "extensions": [
    { "extend":"Foo", "elements":{
      "bar": { "type": "cds.String" }
    }}
  ]
}
```

The same model in **YAML**:

```yaml
requires:
  - @sap/cds/common
  - ./db/schema
definitions:
  some.type: {type: cds.String, length: 11}
  another.type: {type: some.type }
  structured.type:
    elements:
      foo: {type: cds.Integer}
      bar: {type: cds.String}
extensions: [
  - extend: Foo
    elements:
      bar: {type: cds.String}
]
```

The same model as a **plain JavaScript** object:

```js
({
  requires:[ '@sap/cds/common', './db/schema' ],
  definitions: {
    'some.type': { type:"cds.String", length:11 },
    'another.type': { type:"some.type" },
    'structured.type': { elements: {
      'foo': { type:"cds.Integer" },
      'bar': { type:"cds.String" }
    }}
  },
  extensions: [
    { extend:'Foo', elements:{
      'bar': { type:"cds.String" }
    }
  ],
})
```

For the remainder of this spec, you see examples in plain JavaScript representation with the following **conventions**:

```js
({property:...})   // a CSN-specified property name
({'name':...})     // a definition's declared name
"value"            // a string value, including referred names
11, true           // number and boolean literal values
```

Properties

- `requires` – an array listing imported models
- `definitions` – a dictionary of named definitions
- `extensions` – an array of unnamed aspects
- `i18n` – a dictionary of dictionaries of text translations

All properties are optional

For example, one model could contain a few definitions, while another one only contains some extensions.

References are case-sensitive

All references in properties like `type` or `target` use exactly the same notation regarding casing as their targets' names. To avoid problems when translating models to case-insensitive environments like SQL databases, avoid case-significant names and references. For example, avoid two different definitions in the same scope whose names only differ in casing, such as `foo` and `Foo` .

Literals

There are several places where literals can show up in models, such as in SQL expressions, calculated fields, or annotations.

Standard literals are represented as in JSON:

Kind	Example
Globals	`true` , `false` , `null`
Numbers[1]	`11` or `2.4`
Strings	`"foo"`
Dates[2]	`"2016-11-24"`
Times[2]	`"16:11Z"`

Kind	Example
DateTimes[2]	`"2016-11-24T16:11Z"`
Records	`{"foo":<literal>, ...}`
Arrays	`[<literal>, ...]`

In addition, CSN specifies these special forms for references, expressions, and `enum` symbols:

Kind	Example
Unparsed Expressions	`{"=":"foo.bar < 9"}`
Enum symbols[3]	`{"#":"asc"}`

Remarks

> [1] This is as in JSON and shares the same issues when decimals are mapped to doubles with potential rounding errors. The same applies to Integer64. Use strings to avoid that, if applicable.
> [2] Also, as in JSON, dates, and times are represented just as strings as specified in ISO 8601 ; consumers are assumed to know the types and handle the values correctly.
> [3] As enum symbols are equal to their values, it frequently suffices to just provide them as strings. Similar to time and dates in CSN and JSON, the consumers are assumed to know the types and handle the values correctly. The `{"#":...}` syntax option is to serve cases where you have to distinguish the kind only based on the provided value, for example, in untyped annotations.

Definitions

Each entry in the `definitions` dictionary is essentially a type definition. The name is the absolute, fully qualified name of the definition, and the value is a record with the definition details.

Example

```js
({definitions:{
  'Name':     {type:"cds.String"},
  'Currency': {type:"cds.String", length:3},
  'USD':      {type:"Currency"},
  'Amount':   {elements:{
    'value':    {type:"cds.Decimal", precision:11, scale:3},
    'currency': {type:"Currency"},
  }},
  'SortOrder':{enum:{ 'asc':{}, 'desc':{} }}
}})
```

The **name** of a definition is its key in the enclosing dictionary, like in `definitions` for top-level entries or in `elements` for structured types and entities.

Names **must**:

- Be nonempty strings.
- Neither start, nor end with `.` or `::`.
- Not contain substrings `..` or `:::`.
- Not contain the substring `::` more than once.

Properties

- `kind` – one of `context`, `service`, `entity`, `type`, `action`, `function`, or `annotation`
- `type` – an optional base type that this definition is derived from
- `elements` – optional dictionary of *elements* in case of structured types

Property `kind` is always omitted for elements and can be omitted for top-level type definitions. These examples are semantically equivalent:

```js
Foo1 = { type:"cds.String" }
Foo2 = { type:"cds.String", kind:"type" }
```

Type Definitions

Custom-defined types are entries in `definitions` with an optional property `kind = "type"` and the following properties.

Property	Used for
`type`	Scalar Types, Structured Types, and Associations
`elements`	Structured Types
`items`	Arrayed Types
`enum`	Enumeration Types

Example

```js
({definitions: {
  'scalar.type':  {type:"cds.String", length:3 },
  'struct.type':  {elements:{'foo': {type:"cds.Integer"}}},
  'arrayed.type': {items:{type:"cds.Integer"}},
  'enum.type':    {enum:{ 'asc':{}, 'desc':{} }}
}})
```

Properties

- `kind` – omitted or `"type"`
- `type` – the base type, this definition is derived from
- `elements` – optional element definitions for *structured types*.
- `items` – optional definition of item types for *arrayed types*.
- `enum` – an optional dictionary of enum members for *enumeration types*.
- `value` – a constant literal value or calculation expression
- `default` – a default value or expression
- `localized` = *true* if this type was declared like *foo : localized String*
- `...` – other type-specific properties, for example, a String's `length`

Scalar Types

Scalar types always have property `type` specified, plus optional type-specific parameter properties.

```js
({definitions:{
  'scalar.type': {type:"cds.String", length:3 },
}})
```

See the CDL reference docs for an overview of CDS' built-in types.

While in CDS sources you can refer to these types without prefix, they always have to be specified with their **fully qualified names in CSN**, for example:

```js
({definitions: {
  'Foo': { type:"cds.Integer" },
  'Bar': { type:"cds.Decimal", precision:11, scale:3 },
}})
```

Structured Types

Structured types are signified by the presence of an `elements` property. The value of `elements` is a dictionary of `elements` . The name is the local name of the element and the values in turn are Type Definitions.

The optional property `includes` contains a list of fully qualified entity-, aspect-, or type-names. Elements, actions, and annotations from those definitions are then copied into the structured type.

```js
({definitions:{
  'structured.type': {elements:{
    'foo': {type:"cds.Integer"},
    'bar': {type:"cds.String"}
  }}
}})
```

Arrayed Types

Arrayed types are signified by the presence of a property `items` . The value of which is in turn a type definition that specifies the arrayed items' type.

```js
({definitions:{
  'arrayed.type': {items:{type:"cds.Integer"}}
}})
```

Enumeration Types

The `enum` property is a dictionary of enum member elements with the name being the enum symbol and the value being a CQN literal value expression. The literal expression optionally specifies a constant `val` as a literal plus optional annotations. An enumeration type can specify an explicit `type` (for example, *Decimal*) but can also omit it and refer from given enumeration values, or *String* as default.

```js
({definitions:{
  'Gender': {enum:{
    'male':{},
    'female':{},
    'non_binary': {
          val: 'non-binary'
        }
  }},
  'Status': {enum:{
    'submitted': {val:1},
    'fulfilled': {val:2}
  }},
  'Rating': {type:"cds.Decimal", enum:{
    'low':    {val:0},
    'medium': {val:50},
    'high':   {val:100}
  }}
}})
```

Entity Definitions

Entities are structured types with ***kind*** = `'entity'`. In addition, one or more elements usually have property `key` set to true, to flag the entity's primary key.

Example

```js
({definitions:{
  'Products': {kind:"entity", elements:{
    'ID':     {type:"cds.Integer", key:true},
    'title':  {type:"cds.String", notNull:true},
    'price':  {type:"Amount", virtual:true},
  }}
}})
```

Properties

- `kind` – is always `"entity"`
- `elements` – as in Structured Types, optionally equipped with one or more of these boolean properties:
  - `key` – signifies that the element is (part of) the primary key
  - `virtual` – has this element ignored in generic persistence mapping
  - `notNull` – the *not null* constraint as in SQL
- `includes` – as in Structured Types

View Definitions

Views are entities defined as projections on underlying entities. In CSN, views are signified by the presence of property `query`, which captures the projection as a CQN expression.

Example

```js
({definitions:{
  'Foo': { kind:"entity", query: {
    SELECT:{
      from: {ref:['Bar']},
      columns: [ {ref:['title']}, {ref:['price']} ]
    }
  }}
}})
```

Properties

- *kind* – mandatory; always *"entity"*
- *query* – the parsed query in CQN format
- *elements* – optional elements signature, omitted and inferred
- *params* – optional parameters

Views with Declared Signatures

Views with declared signatures have the additional property *elements* filled in as in entities:

```js
({definitions:{
  'with.declared.signature': {kind:"entity",
    elements: {
      'title': {type:"cds.String"},
      'price': {type:"Amount"}
    },
    query: { SELECT:{...} },
  }
}})
```

Views with Parameters

Views with parameters have an additional property *params* – an optional dictionary of parameter type definitions:

```js
({definitions:{
  'with.params': {kind:"entity",
    params: { 'ID': { type: 'cds.Integer' } },
    query: { SELECT:{...} },
  }
}})
```

Projections

Use the `projection` property for views if you don't need the full power of SQL. See `as projection on` in CDL for restrictions.

```js
({ definitions: {
  'Foo': { kind: "entity",
    projection: {
      from: { ref: ['Bar'] },
      columns: [ '*' ]
    }
  }
}})
```

Properties

- `kind` – mandatory; always `"entity"`
- `projection` – the parsed query; equivalent to `query.SELECT`, see CQN
- `elements` – optional elements signature, omitted and inferred

Associations

Associations are like scalar type definitions with `type` being `cds.Association` or `cds.Composition` plus additional properties specifying the association's `target` and optional information like `on` conditions or foreign `keys`.

Basic to-one Associations

The basic form of associations are *to-one* associations to a designated target:

```js
({definitions:{
  'Books': { kind:"entity", elements:{
    'author': { type:"cds.Association", target:"Authors" },
  }},
  //> an association type-def
```

```
    'Currency': { type:"cds.Association", target:"Currencies" },
  }})
```

With Specified *cardinality*

Add property *cardinality* to explicitly specify a *to-one* or *to-many* relationship:

```js
({definitions:{
  'Authors': { kind:"entity", elements:{
    'books': { type:"cds.Association", target:"Books", cardinality:{max:"*"} 
  }},
}})
```

Property *cardinality* is an object *{src?,min?,max}* with...

- *src* set to *1* give a hint to database optimizers, that a source entity always exists
- *min* specifying the target's minimum cardinality – default: *0*
- *max* specifying the target's maximum cardinality – default: *1*

In summary, the default cardinality is *[0..1]*, which means *to-one*.

With Specified *on* Condition

So-called *unmanaged* associations have an explicitly specified *on* condition:

```js
({definitions:{
  'Authors': { kind:"entity", elements:{
    'books': { type:"cds.Association", target:"Books", cardinality{max:"*"},
      on: [{ref:['books', 'author']}, '=', {ref:['$self']}]
    },
  }}
 }})
```

With Specified *keys*

Managed to-one associations automatically use the target's designated primary `key` elements. You can overrule this by explicitly specifying alternative target properties to be used in the `keys` property:

```js
({definitions:{
  'Books': {kind:"entity", elements:{
    'genre': {type:"cds.Association", target:"Genres", keys:[
      {ref:["category"], as:"cat"},
      {ref:["name"]},
    ]},
  }},
}})
```

Property `keys` has the format and mechanisms of CQN projections.

Annotations

Annotations are represented as properties, prefixed with `@`. This format applies to type/entity-level annotations as well as to element-level ones.

Example

```js
({definitions:{
  'Employees': {kind:"entity",
    '@title':"Mitarbeiter",
    '@readonly':true,
    elements:{
      'firstname': {type:"cds.String", '@title':"Vorname"},
      'surname':   {type:"cds.String", '@title':"Nachname"},
    }
  },
}})
```

Annotations are used to add custom information to definitions, the prefixed `@` acts as a protection against conflicts with built-in/standard properties. They're flat lists of key-value pairs, with keys being fully qualified property names and values being represented as introduced in the section Literals and Expressions.

Aspects

In parsed-only models, the top-level property *extensions* holds an array of unapplied extensions or annotations (→ see also Aspects in CDL). The entries are of this form:

```js
ext = { extend|annotate: <name>, <property>: <value>, … }
```

with:

- *extend* or *annotate* referring to the definition to be extended or annotated
- *<property>* being the property that should be extended, for example, *elements* if an entity should be extended with further elements

Extend with <named aspect>

The most basic form allows to express an extension of a named definition with another named definition (→ see Named Aspects):

```js
csn = { extensions:[
  { extend:"TargetDefinition", includes:["NamedAspect"]}
]}
```

Extend with <anonymous aspect>

The form *{ extend:<target>, <property>: <value>, … }* allows to add elements to an existing struct definition as well as to add or override annotations of the target definition:

```js
csn = { extensions:[

  // extend Foo with @foo { ..., bar: String; }
  {
    extend: "Foo",
    '@foo': true,
    elements: {
      // adds a new element 'bar'
      bar: { type: "cds.String", '@bar': true },
    }
  },

]}
```

annotate with <anonymous aspect>

The form `{ annotate:<target>, <property>: <value>, … }` allows to add or override annotations of the target definition as well as those of nested elements:

js
```js
csn = {extensions:[

  // annotate Foo with @foo;
  { annotate:"Foo", '@foo':true },

  // annotate Foo with @foo { boo @boo }
  { annotate:"Foo", '@foo':true, elements: {
    // annotates existing element 'boo'
    boo: {'@boo':true },
  }},

]}
```

Services

Services are definitions with *kind* = `'service'` :

js
```js
({definitions:{
  'MyOrders': {kind:"service"}
}})
```

Actions / Functions

Entity definitions (for *bound* actions/functions) can have an additional property `actions`. The keys of these `actions` are the (local) names of actions/functions. *Unbound* actions/functions of a service are represented as top level definitions.

Example:

```js
({definitions:{
  'OrderService': {kind:"service"},
  'OrderService.Orders': {kind:"entity", elements:{...}, actions:{
    'validate': {kind:"function",
      returns: {type: "cds.Boolean"}
    }
  }},
  'OrderService.cancelOrder': {kind:"action",
      params:{
        'orderID': {type:"cds.Integer"},
        'reason':  {type:"cds.String"},
      },
      returns: {elements:{
        'ack': {enum:{ 'succeeded':{}, 'failed':{} }},
        'msg': {type:"cds.String"},
      }}
    }
  }}
}})
```

Properties

- *`kind`* – either *`"action"`* or *`"function"`* as in *OData*
- *`params`* – a dictionary with the values being Type Definitions
- *`returns`* – a Type Definition describing the response

> Note: The definition of the response can be a reference to a declared type or the inline definition of a new (structured) type.

Imports

The `requires` property lists other models to import definitions from. It is the CSN equivalent of the CDL `using` directive.

Example

```js
({
  requires: [ '@sap/cds/common', './db/schema' ],
  // [...]
})
```

As in Node.js the filenames are either absolute module names or relative filenames, starting with `./` or `../`.

i18n

A CSN may optionally contain a top-level `i18n` property, which can contain translated texts. The expected structure is as follows:

```js
({
  i18n: {
    'language-key': {
      'text-key': "some string"
    }
  }
})
```

This data must be written and handled by the application, there's no out-of-the-box support for this by CAP.