Providing Services

This guide introduces how to define and implement services, leveraging generic implementations provided by the CAP runtimes, complemented by domain-specific custom logic.

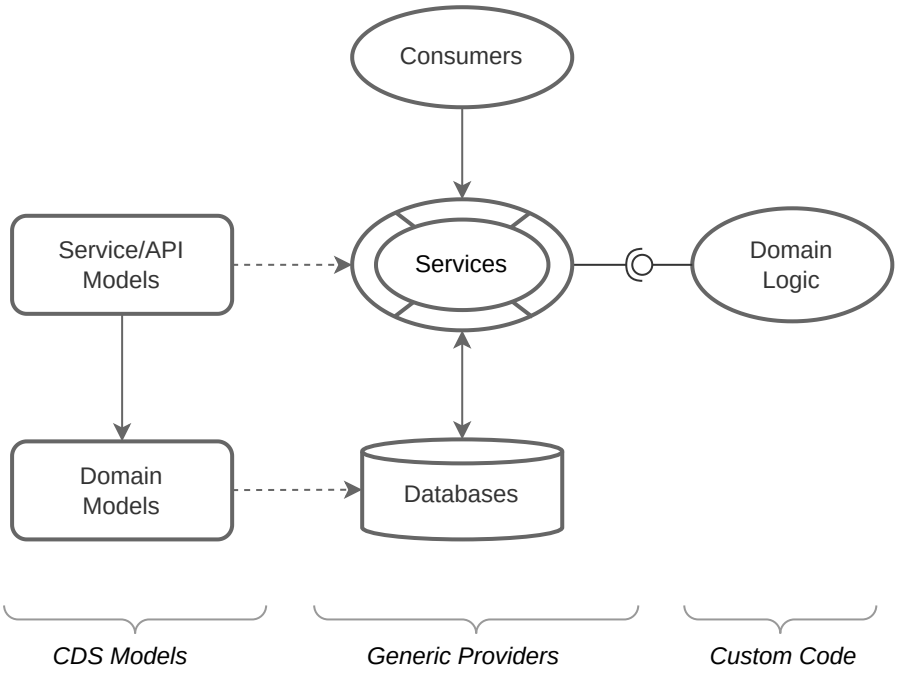
Table of Contents

Intro: Core Concepts

The following sections give a brief overview of CAP's core concepts.

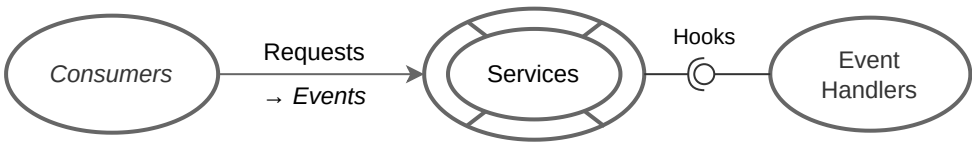
Service-Centric Paradigm

A CAP application commonly provides services defined in CDS models and served by the CAP runtimes. Every active thing in CAP is a service. They embody the behavioral aspects of a domain in terms of exposed entities, actions, and events.



Ubiquitous Events

At runtime, everything happening is in response to events. CAP features a ubiquitous notion of events, which represent both, *requests* coming in through **synchronous** APIs, as well as **asynchronous** *event messages*, blurring the line between both worlds.



Event Handlers

Service providers basically react on events in event handlers, plugged in to respective hooks provided by the core service runtimes.

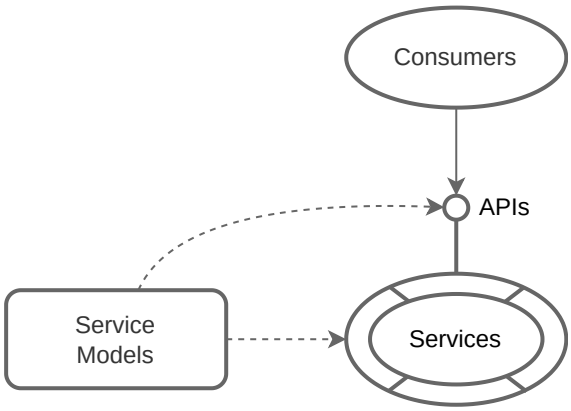
Service Definitions

Services as APIs

In its most basic form, a service definition simply declares the data entities and operations it serves. For example:

```cds
service BookshopService {

  entity Books {
    key ID : UUID;
    title  : String;
    author : Association to Authors;
  }

  entity Authors {
    key ID : UUID;
    name   : String;
    books  : Association to many Books on books.author = $self;
  }

  action submitOrder (book : Books:ID, quantity : Integer);

}
```

This definition effectively defines the API served by *`BookshopService`* .



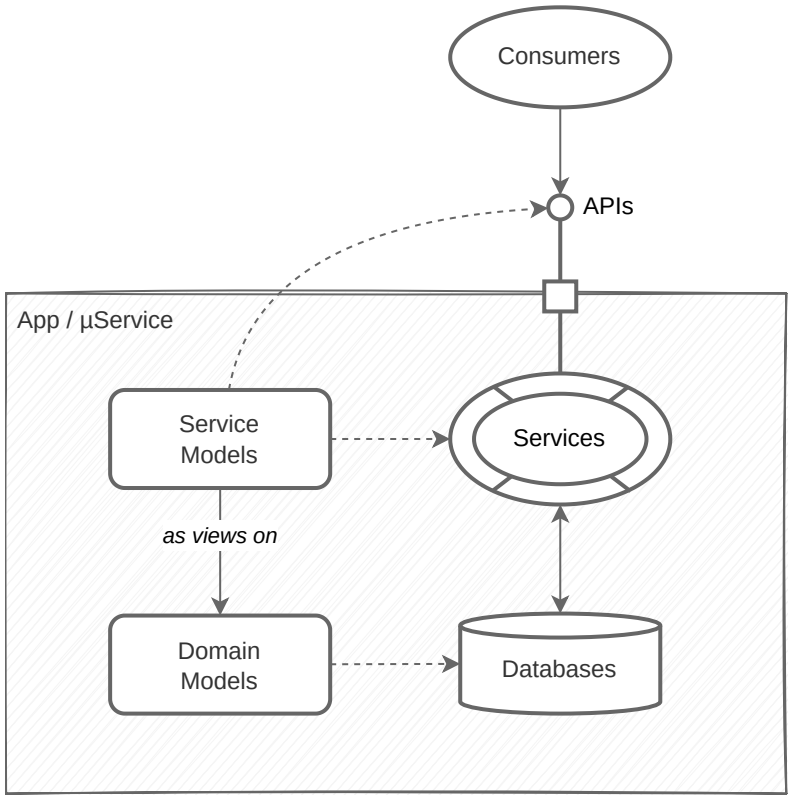
Simple service definitions like that are all we need to run full-fledged servers out of the box, served by CAP's generic runtimes, without any implementation coding required.

Services as Facades

In contrast to the all-in-one definition above, services usually expose views, aka projections, on underlying domain model entities:

```cds
using { sap.capire.bookshop as my } from '../db/schema';
service BookshopService {
  entity Books as projection on my.Books;
  entity Authors as projection on my.Authors;
  action submitOrder (book : Books:ID, quantity : Integer);
}
```

This way, services become facades to encapsulated domain data, exposing different aspects tailored to respective use cases.



Denormalized Views

Instead of exposing access to underlying data in a 1:1 fashion, services frequently expose denormalized views, tailored to specific use cases.

For example, the following service definition, undiscloses information about maintainers from end users and also marks the entities as `@readonly` :

```cds
using { sap.capire.bookshop as my } from '../db/schema';

/** For serving end users */
service CatalogService @(path:'/browse') {

  /** For displaying lists of Books */
  @readonly entity ListOfBooks as projection on Books
  excluding { descr };

  /** For display in details pages */
  @readonly entity Books as projection on my.Books { *,
    author.name as author
  } excluding { createdBy, modifiedBy };

}
```

↳ *Learn more about **CQL** the language used for `projections`.*

↳ *See also: Prefer Single-Purposed Services!*

↳ *Find above sources in **capire/bookshop**.*

Auto-Exposed Entities

Annotate entities with `@cds.autoexpose` to automatically include them in services containing entities with Association referencing to them. For example, this is commonly done for code list entities in order to serve Value Lists dropdowns on UIs:

```cds
service Zoo {
  entity Foo { //...
    code : Association to SomeCodeList;
  }
}
@cds.autoexpose entity SomeCodeList {...}
```

↳ *Learn more about Auto-Exposed Entities in the CDS reference docs.*

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
}
```

↳ *Learn more about Redirected Associations in the CDS reference docs.*

Generic Providers

The CAP runtimes for Node.js and Java provide a wealth of generic implementations, which serve most requests automatically, with out-of-the-box solutions to recurring tasks such as search, pagination, or input validation — the majority of this guide focuses on these generic features.

In effect, a service definition as introduced above is all we need to run a full-fledged server out of the box. The need for coding reduces to real custom logic specific to a project's domain → section Custom Logic picks that up.

Serving CRUD Requests

The CAP runtimes for Node.js and Java provide generic handlers, which automatically serve all CRUD requests to entities for CDS-modelled services on top of a default primary database.

This comprises read and write operations like that:

- *GET /Books/201* → reading single data entities
- *GET /Books?...* → reading data entity sets with advanced query options
- *POST /Books {....}* → creating new data entities
- *PUT/PATCH /Books/201 {...}* → updating data entities
- *DELETE /Books/201* → deleting data entities

> **No filtering and sorting for virtual elements**
>
> CAP runtimes delegate filtering and sorting to the database. Therefore filtering and sorting is not available for `virtual` elements.

Deep Reads and Writes

CDS and the runtimes have advanced support for modeling and serving document-oriented data. The runtimes provide generic handlers for serving deeply nested document structures out of the box as documented in here.

Deep *READ*

You can read deeply nested documents by *expanding* along associations or compositions. For example, like this in OData:

http cds.ql

```http
GET .../Orders?$expand=header($expand=items)
```

↳ *Learn more about `cds.ql`*

Both would return an array of nested structures as follows:

```js
[{
    ID:1, title: 'first order', header: { // to-one
      ID:2, status: 'open', items: [{     // to-many
        ID:3, description: 'first order item'
      },{
        ID:4, description: 'second order item'
      }]
    }
  },
  ...
]
```

Deep *INSERT*

Create a parent entity along with child entities in a single operation, for example, like that:

http

```
POST .../Orders {
  ID:1, title: 'new order', header: { // to-one
    ID:2, status: 'open', items: [{   // to-many
      ID:3, description: 'child of child entity'
    },{
      ID:4, description: 'another child of child entity'
    }]
  }
}
```

Note that Associations and Compositions are handled differently in (deep) inserts and updates:

- Compositions → runtime **deeply creates or updates** entries in target entities
- Associations → runtime **fills in foreign keys** to *existing* target entries

For example, the following request would create a new `Book` with a *reference* to an existing `Author`, with `{ID:12}` being the foreign key value filled in for association `author`:

http

```
POST .../Books {
  ID:121, title: 'Jane Eyre', author: {ID:12}
}
```

Deep `UPDATE`

Deep `UPDATE` of the deeply nested documents look very similar to deep `INSERT`:

http

```
PUT .../Orders/1 {
  title: 'changed title of existing order', header: {
    ID:2, items: [{
      ID:3, description: 'modified child of child entity'
    },{
      ID:5, description: 'new child of child entity'
    }]
  }]
}
```

Depending on existing data, child entities will be created, updated, or deleted as follows:

- entries existing on the database, but not in the payload, are deleted → for example, *ID:4*
- entries existing on the database, and in the payload are updated → for example, *ID:3*
- entries not existing on the database are created → for example, *ID:5*

PUT* vs *PATCH — Omitted fields get reset to *default* values or *null* in case of *PUT* requests; they are left untouched for *PATCH* requests.

Omitted compositions have no effect, whether during *PATCH* or during *PUT*. That is, to delete all children, the payload must specify *null* or *[]*, respectively, for the to-one or to-many composition.

Deep *DELETE*

Deleting a root of a composition hierarchy results in a cascaded delete of all nested children.

sql

```sql
DELETE .../Orders/1  -- would also delete all headers and items
```

Limitations

Note that deep *WRITE* operations are only supported out of the box if the following conditions are met:

1. The on-condition of the composition only uses comparison predicates with an `=` operator.
2. The predicates are only connected with the logical operator *AND*.
3. The operands are references or *$self*. CAP Java also supports pseudo variables like *$user.locale*.

cds

```
entity Orders {
  key ID : UUID;
  title  : String;
- Items  : Composition of many OrderItems on substring(title, 0, 1) <= 'F' or
+ Items  : Composition of many OrderItems on Items.order = $self;
}
```

```
entity OrderItems {
  key order : Association to Orders;
  key pos  : Integer;
  descr: String;
}
```

Auto-Generated Keys

On `CREATE` operations, `key` elements of type `UUID` are filled in automatically. In addition, on deep inserts and upserts, respective foreign keys of newly created nested objects are filled in accordingly.

For example, given a model like that:

cds

```
entity Orders {
  key ID : UUID;
  title  : String;
  Items  : Composition of many OrderItems on Items.order = $self;
}
entity OrderItems {
  key order : Association to Orders;
  key pos  : Integer;
  descr: String;
}
```

When creating a new `Order` with nested `OrderItems` like that:

js

```
POST .../Orders {
  title: 'Order #1', Items: [
    { pos:1, descr: 'Item #1' },
    { pos:2, descr: 'Item #2' }
  ]
}
```

CAP runtimes will automatically fill in `Orders.ID` with a new uuid, as well as the nested `OrderItems.order.ID` referring to the parent.

Searching Data

CAP runtimes provide out-of-the-box support for advanced search of a given text in all textual elements of an entity including nested entities along composition hierarchies.

A typical search request looks like that:

```js
GET .../Books?$search=Heights
```

That would basically search for occurrences of `"Heights"` in all text fields of Books, that is, in `title` and `descr` using database-specific `contains` operations (for example, using `like '%Heights%'` in standard SQL).

The `@cds.search` Annotation

By default search is limited to the elements of type `String` of an entity that aren't calculated or virtual. Yet, sometimes you may want to deviate from this default and specify a different set of searchable elements, or to extend the search to associated entities. Use the `@cds.search` annotation to do so. The general usage is:

```cds
@cds.search: {
    element1,          // included
    element2 : true,   // included
    element3 : false,  // excluded
    assoc1,            // extend to searchable elements in target entity
    assoc2.elementA    // extend to a specific element in target entity
}
entity E { }
```

↳ *Learn more about the syntax of annotations.*

Including Fields

```cds
@cds.search: { title }
entity Books { ... }
```

Searches the `title` element only.

Extend Search to *Associated* Entities

```cds
@cds.search: { author }
entity Books { ... }
```

```cds
@cds.search: { biography: false }
entity Authors { ... }
```

Searches all elements of the `Books` entity, as well as all searchable elements of the associated `Authors` entity. Which elements of the associated entity are searchable is determined by the `@cds.search` annotation on the associated entity. So, from `Authors`, all elements of type `String` are searched but `biography` is excluded.

Extend to Individual Elements in Associated Entities

```cds
@cds.search: { author.name }
entity Books { ... }
```

Searches only in the element `name` of the associated `Authors` entity.

Excluding Fields

```cds
@cds.search: { isbn: false }
entity Books { ... }
```

Searches all elements of type `String` excluding the element `isbn`, which leaves the `title` and `descr` elements to be searched.

> **TIP**
>
> You can explicitly annotate calculated elements to make them searchable, even though they aren't searchable by default. The virtual elements won't be searchable even if they're explicitly annotated.

The `@Common.Text` Annotation

If an entity has an element annotated with the `@Common.Text` annotation, then the property that holds the display text is added to the list of searchable elements (see exception below).

For example, with the following model, the list of searchable elements for `Books` is `title` and `author.name`:

```cds
entity Books : cuid {
  title  : String;
  @Common.Text : author.name
```

```
  author : Association to Author;
}
entity Author : cuid {
  name : String;
}
```

> **`@cds.search` takes precedence over `@Common.Text`**
>
> As a result, `@Common.Text` is ignored as soon as `@cds.search` defines anything in including mode. Only if you exclusively exclude properties using `@cds-search`, the `@Common.Text` is kept.

To illustrate the above:

- `@cds.search: { title: false }` on `Books` would only exclude properties, so `author.name` would still be searched.
- `@cds.search: { title }` on `Books` defines an include list, so `author.name` is not searched. In this mode, `@cds.search` is expected to include all properties that should be searched. Hence, `author.name` would need to be added to `@cds.search` itself: `@cds.search: { title, author.name }`.

Fuzzy Search on SAP HANA Cloud beta

> Prerequisite: For CAP Java, you need to run in `HEX` optimization mode on SAP HANA Cloud and enable `cds.sql.hana.search.fuzzy` ☼

Fuzzy search is a fault-tolerant search feature of SAP HANA Cloud, which returns records even if the search term contains additional characters, is missing characters, or has typographical errors.

You can configure the fuzziness in the range [0.0, 1.0]. The value 1.0 enforces exact search.

- Java: `cds.sql.hana.search.fuzzinessThreshold` ☼
- Node.js: `cds.hana.fuzzy` ☼ [(1)]

[(1)] If set to `false`, fuzzy search is disabled and falls back to a case insensitive substring search.

Override the fuzziness for elements, using the `@Search.fuzzinessThreshold` annotation:

```cds
entity Books {
   @Search.fuzzinessThreshold: 0.7
   title : String;
}
```

The relevance of a search match depends on the weight of the element causing the match. By default, all searchable elements have equal weight. To adjust the weight of an element, use the `@Search.ranking` annotation. Allowed values are HIGH, MEDIUM (default), and LOW:

```cds
entity Books {
   @Search.ranking: HIGH
   title         : String;

   @Search.ranking: LOW
   publisherName : String;
}
```

> **Wildcards in search terms**
>
> When using wildcards in search terms, an *exact pattern search* is performed. Supported wildcards are '*' matching zero or more characters and '?' matching a single character. You can escape wildcards using '\'.

Pagination & Sorting

Implicit Pagination

By default, the generic handlers for READ requests automatically **truncate** result sets to a size of 1,000 records max. If there are more entries available, a link is added to the response allowing clients to fetch the next page of records.

The OData response body for truncated result sets contains a `nextLink` as follows:

```http
GET .../Books
>{
  value: [
    {... first record ...},
    {... second record ...},
    ...
```

```
  ],
  @odata.nextLink: "Books?$skiptoken=1000"
}
```

To retrieve the next page of records from the server, the client would use this `nextLink` in a follow-up request, like so:

```http
GET .../Books?$skiptoken=1000
```

On firing this query, you get the second set of 1,000 records with a link to the next page, and so on, until the last page is returned, with the response not containing a `nextLink` .

> **WARNING**
>
> Per OData specification for Server Side Paging , the value of the `nextLink` returned by the server must not be interpreted or changed by the clients.

Reliable Pagination

> Note: This feature is available only for OData V4 endpoints.

Using a numeric skip token based on the values of `$skip` and `$top` can result in duplicate or missing rows if the entity set is modified between the calls. *Reliable Pagination* avoids this inconsistency by generating a skip token based on the values of the last row of a page.

The reliable pagination is available with following limitations:

- Results of functions or arithmetic expressions can't be used in the `$orderby` option (explicit ordering).
- The elements used in the `$orderby` of the request must be of simple type.
- All elements used in `$orderby` must also be included in the `$select` option, if it's set.
- Complex concatenations of result sets aren't supported.

> **WARNING**
>
> Don't use reliable pagination if an entity set is sorted by elements that contain sensitive information, the skip token could reveal the values of these elements.

The feature can be enabled with the following configuration options set to `true` :

- Java: *`cds.query.limit.reliablePaging.enabled`* ☼
- Node.js: *`cds.query.limit.reliablePaging`* ☼

Paging Limits

You can configure default and maximum page size limits in your project configuration as follows:

```json
"cds": {
  "query": {
    "limit": {
      "default": 20, //> no default
      "max": 100     //> default 1000
    }
  }
}
```

- The **maximum limit** defines the maximum number of items that can get retrieved, regardless of *`$top`* .
- The **default limit** defines the number of items that are retrieved if no *`$top`* was specified.

Annotation *`@cds.query.limit`*

You can override the defaults by applying the *`@cds.query.limit`* annotation on the service or entity level, as follows:

```cds
@cds.query.limit: { default?, max? } | Number
```

The limit definitions for *`CatalogService`* and *`AdminService`* in the following example are equivalent.

```cds
@cds.query.limit.default: 20
@cds.query.limit.max: 100
service CatalogService {
  // ...
}
@cds.query.limit: { default: 20, max: 100 }
service AdminService {
  // ...
}
```

`@cds.query.limit` can be used as shorthand if no default limit needs to be specified at the same level.

```cds
@cds.query.limit: 100
service CatalogService {
  entity Books as projection on my.Books;     //> pages at 100
  @cds.query.limit: 20
  entity Authors as projection on my.Authors; //> pages at 20
}
service AdminService {
  entity Books as projection on my.Books;     //> pages at 1000 (default)
}
```

Precedence

The closest limit applies, that means, an entity-level limit overrides that of its service, and a service-level limit overrides the global setting. The value `0` disables the respective limit at the respective level.

```cds
@cds.query.limit.default: 20
service CatalogService {
  @cds.query.limit.max: 100
  entity Books as projection on my.Books;     //> default = 20 (from CatalogS
  @cds.query.limit: 0
  entity Authors as projection on my.Authors; //> no default, max = 1,000 (fr
}
```

Implicit Sorting

Paging requires implied sorting, otherwise records might be skipped accidentally when reading follow-up pages. By default the entity's primary key is used as a sort criterion.

For example, given a service definition like this:

```cds
service CatalogService {
  entity Books as projection on my.Books;
}
```

The SQL query executed in response to incoming requests to Books will be enhanced with an additional order-by clause as follows:

```sql
SELECT ... from my_Books
ORDER BY ID; -- default: order by the entity's primary key
```

If the request specifies a sort order, for example, `GET .../Books?$orderby=author`, both are applied as follows:

```sql
SELECT ... from my_Books ORDER BY
  author,     -- request-specific order has precedence
  ID;         -- default order still applied in addition
```

We can also define a default order when serving books as follows:

```cds
service CatalogService {
  entity Books as projection on my.Books order by title asc;
}
```

Now, the resulting order by clauses are as follows for `GET .../Books`:

```sql
SELECT ... from my_Books ORDER BY
  title asc,  -- from entity definition
  ID;         -- default order still applied in addition
```

... and for `GET .../Books?$orderby=author`:

```sql
SELECT ... from my_Books ORDER BY
  author,     -- request-specific order has precedence
  title asc,  -- from entity definition
  ID;         -- default order still applied in addition
```

Concurrency Control

CAP runtimes support different ways to avoid lost-update situations as documented in the following.

Use *optimistic locking* to *detect* concurrent modification of data *across requests*. The implementation relies on ETags.

Use *pessimistic locking* to *protect* data from concurrent modification by concurrent *transactions*. CAP leverages database locks for pessimistic locking.

Conflict Detection Using ETags

The CAP runtimes support optimistic concurrency control and caching techniques using ETags. An ETag identifies a specific version of a resource found at a URL.

Enable ETags by adding the *`@odata.etag`* annotation to an element to be used to calculate an ETag value as follows:

```cds
using { managed } from '@sap/cds/common';
entity Foo : managed {...}
annotate Foo with { modifiedAt @odata.etag }
```

> The value of an ETag element should uniquely change with each update per row. The *`modifiedAt`* element from the pre-defined *`managed`* aspect is a good candidate, as this is automatically updated. You could also use update counters or UUIDs, which are recalculated on each update.

You use ETags when updating, deleting, or invoking the action bound to an entity by using the ETag value in an *`If-Match`* or *`If-None-Match`* header. The following examples represent typical requests and responses:

```http
POST Employees { ID:111, name:'Name' }
> 201 Created {'@odata.etag': 'W/"2000-01-01T01:10:10.100Z"',...}
//> Got new ETag to be used for subsequent requests...
```

```http
GET Employees/111
If-None-Match: "2000-01-01T01:10:10.100Z"
> 304 Not Modified // Record was not changed
```

```http
GET Employees/111
If-Match: "2000-01-01T01:10:10.100Z"
> 412 Precondition Failed // Record was changed by another user
```

```http
UPDATE Employees/111
If-Match: "2000-01-01T01:10:10.100Z"
> 200 Ok {'@odata.etag': 'W/"2000-02-02T02:20:20.200Z"',...}
//> Got new ETag to be used for subsequent requests...
```

```http
UPDATE Employees/111
If-Match: "2000-02-02T02:20:20.200Z"
> 412 Precondition Failed // Record was modified by another user
```

```http
DELETE Employees/111
If-Match: "2000-02-02T02:20:20.200Z"
> 412 Precondition Failed // Record was modified by another user
```

If the ETag validation detects a conflict, the request typically needs to be retried by the client. Hence, optimistic concurrency should be used if conflicts occur rarely.

Pessimistic Locking

Pessimistic locking allows you to lock the selected records so that other transactions are blocked from changing the records in any way.

Use *exclusive* locks when reading entity data with the *intention to update* it in the same transaction and you want to prevent the data to be locked or updated in a concurrent transaction.

Use *shared* locks if you only need to prevent the entity data to be locked exclusively by an update in a concurrent transaction or by a read operation with lock mode *exclusive*. Non-locking read operations or read operations with lock mode *shared* are not prevented.

The records are locked until the end of the transaction by commit or rollback statement.

Here's an overview table:

State	Select Without Lock	Select With Shared Lock	Select With Exclusive Lock/Update
not locked	passes	passes	passes
shared lock	passes	passes	waits
exclusive lock	passes	waits	waits

↳ *Learn more about using the `SELECT ... FOR UPDATE` statement in the Node.js runtime.*

↳ *Learn more about using the `Select.lock()` method in the Java runtime.*

> **Restrictions**
>
> - Pessimistic locking is supported for domain entities (DB table rows). The locking is not possible for projections and views.
> - Pessimistic locking is not supported by SQLite. H2 supports exclusive locks only.

Input Validation

CAP runtimes automatically validate user input, controlled by the following annotations.

`@readonly`

Elements annotated with `@readonly`, as well as *calculated elements*, are protected against write operations. That is, if a CREATE or UPDATE operation specifies values for such fields, these values are **silently ignored**.

By default `virtual` elements are also *calculated*.

> **TIP**
>
> The same applies for fields with the OData Annotations `@FieldControl.ReadOnly` (static), `@Core.Computed`, or `@Core.Immutable` (the latter only on UPDATEs).

> **Not allowed on keys**
>
> Do not use the `@readonly` annotation on keys in all variants.

`@mandatory`

Elements marked with `@mandatory` are checked for nonempty input: `null` and (trimmed) empty strings are rejected.

```cds
service Sue {
  entity Books {
    key ID : UUID;
    title  : String @mandatory;
  }
}
```

In addition to server-side input validation as introduced above, this adds a corresponding `@FieldControl` annotation to the EDMX so that OData / Fiori clients would enforce a valid entry, thereby avoiding unnecessary request roundtrips:

```xml
<Annotations Target="Sue.Books/title">
  <Annotation Term="Common.FieldControl" EnumMember="Common.FieldControlType/I
</Annotations>
```

@assert .format

Allows you to specify a regular expression string (in ECMA 262 format in CAP Node.js and java.util.regex.Pattern format in CAP Java) that all string input must match.

```cds
entity Foo {
  bar : String @assert.format: '[a-z]ear';
}
```

@assert .range

Allows you to specify *`[ min, max ]`* ranges for elements with ordinal types — that is, numeric or date/time types. For *`enum`* elements, *`true`* can be specified to restrict all input to the defined enum values.

```cds
entity Foo {
  bar : Integer  @assert.range: [ 0, 3 ];
  boo : Decimal  @assert.range: [ 2.1, 10.25 ];
  car : DateTime @assert.range: ['2018-10-31', '2019-01-15'];
  zoo : String   @assert.range enum { high; medium; low; };
}
```

... with open intervals

By default, specified *`[min,max]`* ranges are interpreted as closed intervals, that means, the performed checks are *`min ≤ input ≤ max`* . You can also specify open intervals by wrapping the *min* and/or *max* values into parentheses like that:

```cds
@assert.range: [(0),100]    // 0 < input ≤ 100
@assert.range: [0,(100)]    // 0 ≤ input < 100
@assert.range: [(0),(100)]  // 0 < input < 100
```

In addition, you can use an underscore `_` to represent *Infinity* like that:

```cds
@assert.range: [(0),_]  // positive numbers only, _ means +Infinity here
@assert.range: [_,(0)]  // negative number only, _ means -Infinity here
```

> Basically values wrapped in parentheses `(x)` can be read as *excluding* `x` for *min* or *max*. Note that the underscore `_` doesn't have to be wrapped into parentheses, as by definition no number can be equal to *Infinity* .

Support for open intervals and infinity is available for CAP Node.js since `@sap/cds` version **8.5** and in CAP Java since version **3.5.0**.

`@assert .target`

Annotate a managed to-one association of a CDS model entity definition with the `@assert.target` annotation to check whether the target entity referenced by the association (the reference's target) exists. In other words, use this annotation to check whether a non-null foreign key input in a table has a corresponding primary key in the associated/referenced target table.

You can check whether multiple targets exist in the same transaction. For example, in the `Books` entity, you could annotate one or more managed to-one associations with the `@assert.target` annotation. However, it is assumed that dependent values were inserted before the current transaction. For example, in a deep create scenario, when creating a book, checking whether an associated author exists that was created as part of the same deep create transaction isn't supported, in this case, you will get an error.

The `@assert.target` check constraint is meant to **validate user input** and not to ensure referential integrity. Therefore only `CREATE` , and `UPDATE` events are supported (`DELETE` events are not supported). To ensure that every non-null foreign key in a table has a corresponding primary key in the associated/referenced target table (ensure referential integrity), the `@assert.integrity` constraint must be used instead.

If the reference's target doesn't exist, an HTTP response (error message) is provided to HTTP client applications and logged to stdout in debug mode. The HTTP response body's content adheres to the standard OData specification for an error response body .

Example

Add `@assert.target` annotation to the service definition as previously mentioned:

```cds
entity Books {
  key ID : UUID;
  title  : String;
  author : Association to Authors @assert.target;
}

entity Authors {
  key ID : UUID;
  name   : String;
  books  : Association to many Books on books.author = $self;
}
```

HTTP Request — *assume that an author with the ID `"796e274a-c3de-4584-9de2-3ffd7d42d646"` doesn't exist in the database*

```http
POST Books HTTP/1.1
Accept: application/json;odata.metadata=minimal
Prefer: return=minimal
Content-Type: application/json;charset=UTF-8

{"author_ID": "796e274a-c3de-4584-9de2-3ffd7d42d646"}
```

HTTP Response

```http
HTTP/1.1 400 Bad Request
odata-version: 4.0
content-type: application/json;odata.metadata=minimal

{"error": {
  "@Common.numericSeverity": 4,
  "code": "400",
  "message": "Value doesn't exist",
  "target": "author_ID"
}}
```

> **TIP**
>
> In contrast to the `@assert.integrity` constraint, whose check is performed on the underlying database layer, the `@assert.target` check constraint is performed on the application service layer before the custom application handlers are called.

> **WARNING**
>
> Cross-service checks are not supported. It is expected that the associated entities are defined in the same service.

> **WARNING**
>
> The *`@assert.target`* check constraint relies on database locks to ensure accurate results in concurrent scenarios. However, locking is a database-specific feature, and some databases don't permit to lock certain kinds of objects. On SAP HANA, for example, views with joins or unions can't be locked. Do not use *`@assert.target`* on such artifacts/entities.

Custom Error Messages

The annotations *`@assert.range`* , *`@assert.format`* , and *`@mandatory`* also support custom error messages. Use the annotation *`@<anno>.message`* with an error text or text bundle key to specify a custom error message:

```cds
entity Person : cuid {
  name : String;

  @assert.format: '/^\S+@\S+\.\S+$/'
  @assert.format.message: 'Provide a valid email address'
  email : String;

  @assert.range: [(0),_]
  @assert.range.message: '{i18n>person-age}'
  age : Int16;
}
```

Note: The above can also be written like that:

```cds
entity Person : cuid {
  name : String;

  @assert.format: {
    $value: '/^\S+@\S+\.\S+$/', message: 'Provide a valid email address'
  }
  email : String;
```

```
  @assert.range: {
    $value: [(0),_], message: '{i18n>person-age}'
  }
  age : Int16;
}
```

Database Constraints

Next to input validation, you can add database constraints to prevent invalid data from being persisted.

Custom Logic

As most standard tasks and use cases are covered by generic service providers, the need to add service implementation code is greatly reduced and minified, and hence the quantity of individual boilerplate coding.

The remaining cases that need custom handlers, reduce to real custom logic, specific to your domain and application, such as:

- Domain-specific programmatic Validations
- Augmenting result sets, for example to add computed fields for frontends
- Programmatic Authorization Enforcements
- Triggering follow-up actions, for example calling other services or emitting outbound events in response to inbound events
- And more... In general, all the things not (yet) covered by generic handlers

In Node.js, the easiest way to add custom implementations for services is through equally named *.js* files placed next to a service definition's *.cds* file:

```sh
./srv
  - cat-service.cds  # service definitions
  - cat-service.js   # service implementation
...
```

↳ *Learn more about providing service implementations in Node.js.*

In Java, you'd assign `EventHandler` classes using dependency injection as follows:

```java
@Component
@ServiceName("org.acme.Foo")
public class FooServiceImpl implements EventHandler {...}
```

↳ *Learn more about Event Handler classes in Java.*

Custom Event Handlers

Within your custom implementations, you can register event handlers like that:

Node.js Java

```js
module.exports = function (){
  this.on ('submitOrder', (req)=>{...}) //> custom actions
  this.on ('CREATE',`Books`, (req)=>{...})
  this.before ('UPDATE',`*`, (req)=>{...})
  this.after ('READ',`Books`, (books)=>{...})
}
```

↳ *Learn more about **adding event handlers in Node.js**.*

↳ *Learn more about **adding event handlers in Java**.*

Hooks: `on` , `before` , `after`

In essence, event handlers are functions/method registered to be called when a certain event occurs, with the event being a custom operation, like `submitOrder` , or a CRUD operation on a certain entity, like `READ Books` ; in general following this scheme:

- `<hook:on|before|after>` , `<event>` , `[<entity>]` → handler function

CAP allows to plug in event handlers to these different hooks, that is phases during processing a certain event:

- `on` handlers run *instead of* the generic/default handlers.
- `before` handlers run *before* the `on` handlers

- `after` handlers run *after* the `on` handlers, and get the result set as input

`on` handlers form an *interceptor* stack: the topmost handler getting called by the framework. The implementation of this handler is in control whether to delegate to default handlers down the stack or not.

`before` and `after` handlers are *listeners*: all registered listeners are invoked in parallel. If one vetoes / throws an error the request fails.

Within Event Handlers

Event handlers all get a uniform *Request/Event Message* context object as their primary argument, which, among others, provides access to the following information:

- The `event` name — that is, a CRUD method name, or a custom-defined one
- The `target` entity, if any
- The `query` in CQN format, for CRUD requests
- The `data` payload
- The `user` , if identified/authenticated
- The `tenant` using your SaaS application, if enabled

↳ *Learn more about **implementing event handlers in Node.js**.*
↳ *Learn more about **implementing event handlers in Java**.*

Actions & Functions

In addition to common CRUD operations, you can declare domain-specific custom operations as shown below. These custom operations always need custom implementations in corresponding events handlers.

You can define actions and functions in CDS models like that:

```cds
service Sue {
  // unbound actions & functions
  function sum (x:Integer, y:Integer) returns Integer;
```

```
  function stock (id : Foo:ID) returns Integer;
  action add (x:Integer, to: Integer) returns Integer;

  // bound actions & functions
  entity Foo { key ID:Integer } actions {
    function getStock() returns Integer;
    action order (x:Integer) returns Integer;
    // bound to the collection and not a specific instance of Foo
    action customCreate (in: many $self, x: String) returns Foo;
    // All parameters are optional by default, unless marked with `not null`:
    action discard (reason: String not null);
  }
}
```

↳ *Learn more about modeling actions and functions in CDS.*

The differentiation between *Actions* and *Functions* as well as *bound* and *unbound* stems from the OData specifications, and in essence is as follows:

- **Actions** modify data in the server
- **Functions** retrieve data
- **Unbound** actions/functions are like plain unbound functions in JavaScript.
- **Bound** actions/functions always receive the bound entity's primary key as implicit first argument, similar to `this` pointers in Java or JavaScript. The exception are bound actions to collections, which are bound against the collection and not a specific instance of the entity. An example use case are custom create actions for the SAP Fiori elements UI.

Implementing Actions / Functions

In general, implement actions or functions like that:

```js
module.exports = function Sue(){
  this.on('sum', ({data:{x,y}}) => x+y)
  this.on('add', ({data:{x,to}}) => stocks[to] += x)
  this.on('stock', ({data:{id}}) => stocks[id])
  this.on('getStock','Foo', ({params:[id]}) => stocks[id])
  this.on('order','Foo', ({params:[id],data:{x}}) => stocks[id] -= x)
}
```

Event handlers for actions or functions are very similar to those for CRUD events, with the name of the action/function replacing the name of the CRUD operations. No entity is specific for unbound actions/functions.

Method-style Implementations in Node.js, you can alternatively implement actions and functions using conventional JavaScript methods with subclasses of `cds.Service` :

```js
module.exports = class Sue extends cds.Service {
  sum(x,y) { return x+y }
  add(x,to) { return stocks[to] += x }
  stock(id) { return stocks[id] }
  getStock(Foo,id) { return stocks[id] }
  order(Foo,id,x) { return stocks[id] -= x }
}
```

Calling Actions / Functions

HTTP Requests to call the actions/function declared above look like that:

```js
GET .../sue/sum(x=1,y=2)              // unbound function
GET .../sue/stock(id=2)               // unbound function
POST .../sue/add {"x":11,"to":2}      // unbound action
GET .../sue/Foo(2)/Sue.getStock()     // bound function
POST .../sue/Foo(2)/Sue.order {"x":3} // bound action
```

> Note: You always need to add the `()` for functions, even if no arguments are required. The OData standard specifies that bound actions/functions need to be prefixed with the service's name. In the previous example, entity `Foo` has a bound action `order` . That action must be called via `/Foo(2)/Sue.order` instead of simply `/Foo(2)/order` . For convenience, you may:
>
> - Call bound actions/functions without prefixing them with the service name.
> - Omit the `()` if no parameter is required.
> - Use query options to provide function parameters like `sue/sum?x=1&y=2`

Programmatic usage via **generic APIs** for Node.js:

For unbound actions and functions:

```ts
async function srv.send (
  event    : string | { event, data?, headers?: object },
  data?    : object | any
)
return : result of this.dispatch(req)
```

For bound actions and functions:

```ts
async function srv.send (
 event    : string | { event, entity, data?, params?: array of object, headers
 entity   : string,
 data?    : object | any
)
return : result of this.dispatch(req)
```

- *event* is a name of a custom action or function
- *entity* is a name of an entity
- *params* are keys of the entity instance

Programmatic usage would look like this for Node.js:

```js
const srv = await cds.connect.to('Sue')
// unbound actions/functions
await srv.send('sum',{x:1,y:2})
await srv.send('stock',{id:2})
await srv.send('add',{x:11,to:2})
// actions/functions bound to collection
await srv.send('getStock','Foo',{id:2})
// for actions/functions bound to entity instance, use this syntax
await srv.send({ event: 'order', entity: 'Foo', data: {x:3}, params: [{id:2
```

> Note: Always pass the target entity name as second argument for bound actions/functions.

Programmatic usage via **typed API** — Node.js automatically equips generated service instances with specific methods matching the definitions of actions/functions found in the services' model. This allows convenient usage like that:

```js
const srv = await cds.connect.to(Sue)
// unbound actions/functions
srv.sum(1,2)
srv.stock(2)
srv.add(11,2)
// bound actions/functions
srv.getStock('Foo',2)
srv.order('Foo',2,3)
```

> Note: Even with that typed APIs, always pass the target entity name as second argument for bound actions/functions.

Status-Transition Flows beta

The flow feature makes it easy to define and manage state transitions in your CDS models. It ensures transitions are explicitly modeled, validated, and executed in a controlled and reliable way. For more complex requirements, you can extend flows with custom event handlers.

Enabling Flows

Status-transition flows are supported by both CAP runtimes.

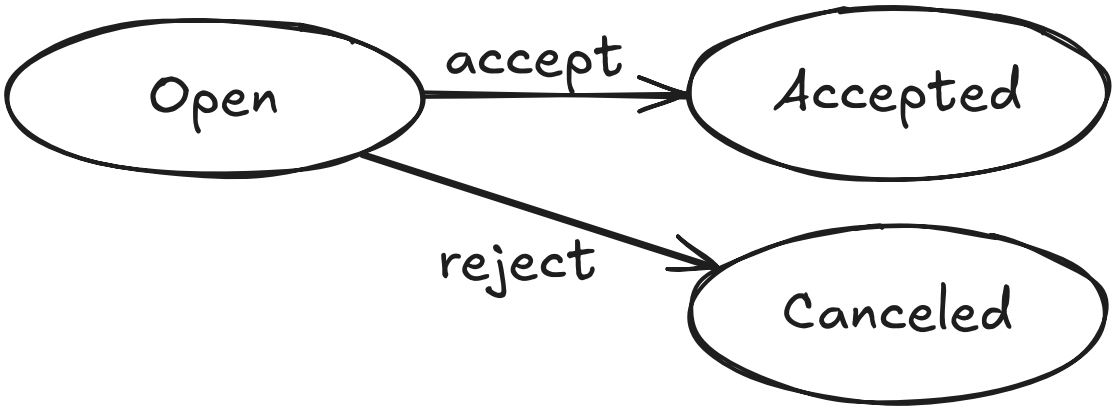
In CAP Node.js support for flows is part of the CAP Node.js core (*@sap/cds*).

For CAP Java, support for flows is provided by the feature cds-feature-flow . Enable it by adding this dependency to your *srv/pom.xml* file:

```xml
<dependency>
  <groupId>com.sap.cds</groupId>
  <artifactId>cds-feature-flow</artifactId>
  <scope>runtime</scope>
</dependency>
```

Modeling Flows

The following example, taken from @capire/xtravels , shows the simplest way to model a flow. The annotations in the service model are sufficient to define and use the flow.



The following is an extract of the relevant parts of the domain model:

▶ *db/schema.cds*

cds

```
// srv/travel-service.cds
service TravelService {

  // Define entity and actions
  entity Travels as projection on db.Travels
  actions {
    action rejectTravel();
    action acceptTravel();
    action deductDiscount( percent: Percentage not null ) returns Travels;
  };

  // Define flow through actions (+ status check for "deductDiscount")
  annotate Travels with @flow.status: Status actions {
    rejectTravel    @from: #Open  @to: #Canceled;
    acceptTravel    @from: #Open  @to: #Accepted;
    deductDiscount  @from: #Open;
  };

}
```

No custom action handlers are needed for simple transitions—the flow feature's default handlers validate that the entry state is `Open` and transition the status to `Accepted` or

`Canceled` accordingly. For more complex scenarios, you can add custom handlers as explained later.

Flow Annotations

Flows consist of a *status element* and a set of *flow actions* that define transitions between states.

Declare Flow Using `@flow.status`

To model a flow, one of the entity fields needs to be annotated with `@flow.status`. This field must be one of the following:

- A String or Integer enum consisting of keys and values
- A String enum with only symbols
- A Codelist entity with the key `code` if localization is needed (`code` must be one of the two above)

> **The status field should be `@readonly` and have a default value.**
>
> We recommend to always use `@flow.status` in combination with `@readonly`. This ensures that the status element is immutable from the client side, giving the service provider full control over all state transitions. As no initial state can be provided on `CREATE`, there should be a default value.

When you annotate `@flow.status: <element name>` at the entity level (as in the example above), the annotation is propagated to the respective element, which is also automatically annotated with `@readonly`.

About the `@flow.status` annotation:

- This annotation is **mandatory**.
- The annotated element must be either an enum or an association to a code list.
- Only one status element per entity is supported.
- Draft-enabled entities are supported, however flows are only applied to the active version.
- `null` is **not** a valid state—model your empty state explicitly.

> **Only simple projections are supported**

> The entity must be *writable*, and renaming the status element is currently not supported.

After declaring `@flow.status`, use the following annotations on bound actions to model transitions:

Model Transitions Using `@from` and `to`

Both annotations are optional, but at least one is required to mark an action as a flow action. Use either one or both depending on your needs. When you use both, no custom handlers are needed—generic handlers are registered automatically.

`@from`

- Defines valid entry states for the action.
- Validates whether the entity is in a valid entry state before executing the action (the current state of the entity must be included in the states defined here).
- Can be a single value or an array of values (each element must be a value from the status enum).
- UI annotations to allow/disallow buttons and to refresh the page are automatically generated for UI5.
  - Can be deactivated via `cds.features.annotate_for_flows: false` ☼.

`@to`

- Defines the desired target state of the entity after executing the action.
- Changes the state of the entity to the value defined in this annotation after executing the action.
- Must be a single value from the status enum.

Generic Handlers

Generic handlers are registered automatically, so no custom implementations are required for basic flows.

`before`

Based on the `@from` annotation, a handler validates that the entity is in a valid entry state - the current state must match one of the states specified in `@from`. If validation fails,

the request returns a `409 Conflict` HTTP status code with an appropriate error message.

on

In case of a `@to` declaration and if no custom handler is provided, an empty handler is registered that completes the action for void return types, ensuring the request passes through the generic handler stack. This is an exception to the rule that actions must be implemented by the application.

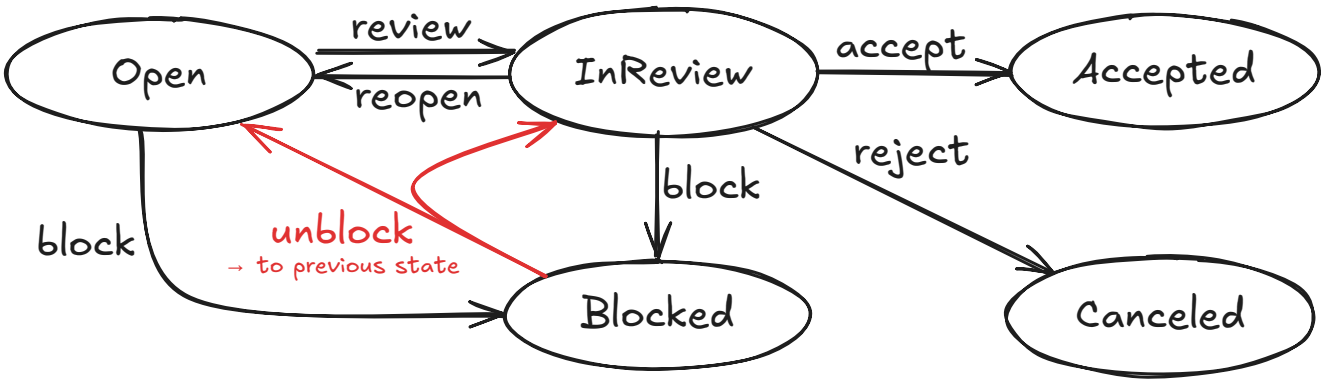
after

Based on the `@to` annotation, a handler automatically updates the entity's status to the target state. For example, if the current state is `Open` and the target state is `Accepted`, the handler updates the status to `Accepted` after action execution. This ensures consistent state transitions without custom logic.

> **Generic handlers are not executed for draft entities**
>
> For example, if you call `acceptTravel()` on a `Travels` entity that is currently being edited (in *inactive* state), the call has no effect.

Reverting to Previous State

You can use the target state `$flow.previous` to restore the previous state in a workflow. The following example introduces a `Blocked` state with two possible previous states (`Open` and `InReview`) and an action `unblockTravel` that restores the previous state. For instance, if `Blocked` was transitioned to from `Open`, calling `unblockTravel` transitions back to `Open`. The same applies for `InReview`.



```cds
// srv/travel-service.cds
service TravelService {

  // Define entity and actions
  entity Travels as projection on db.Travels
  actions {
    action reviewTravel();
    action reopenTravel();
    action blockTravel();
    action unblockTravel();
    action rejectTravel();
    action acceptTravel();
    action deductDiscount( percent: Percentage not null ) returns Travels;
  };

  // Define flow incl. "unblockTravel" that transitions to the previous state
  annotate Travels with @flow.status: Status actions {
    reviewTravel    @from: #Open                @to: #InReview;
    reopenTravel    @from: #InReview            @to: #Open;
    blockTravel     @from: [#Open, #InReview]   @to: #Blocked;
    unblockTravel   @from: #Blocked             @to: $flow.previous;
    rejectTravel    @from: #InReview            @to: #Canceled;
    acceptTravel    @from: #InReview            @to: #Accepted;
    deductDiscount  @from: #Open;
  };

}
```

Entities with flows that include at least one transition to `$flow.previous` are automatically extended with the `sap.common.FlowHistory` aspect, which includes `transitions_` composition that captures the history of state transitions.

▶ *The `transitions_` composition*

Extending Flows

Flow annotations work well for basic flows. For more complex scenarios, implement custom event handlers.

Common use cases for custom handlers:

- **Additional validation:** Implement a custom `before` handler when entry state validation depends on extra conditions
- **Non-void return types:** Implement a custom `on` handler when the action returns data
- **Conditional target states:** Implement a custom `on` or `after` handler (without `@to` annotation) when multiple target states depend on conditions

---

Serving Media Data

CAP provides out-of-the-box support for serving media and other binary data.

Annotating Media Elements

You can use the following annotations in the service model to indicate that an element in an entity contains media data.

`@Core.MediaType` : Indicates that the element contains media data (directly or using a redirect). The value of this annotation is either a string with the contained MIME type (as shown in the first example), or is a path to the element that contains the MIME type (as shown in the second example).

`@Core.IsMediaType` : Indicates that the element contains a MIME type. The `@Core.MediaType` annotation of another element can reference this element.

`@Core.IsURL @Core.MediaType` : Indicates that the element contains a URL pointing to the media data (redirect scenario).

`@Core.ContentDisposition.Filename` : Indicates that the element is expected to be displayed as an attachment, that is downloaded and saved locally. The value of this annotation is a path to the element that contains the Filename (as shown in the fourth example).

`@Core.ContentDisposition.Type` : Can be used to instruct the browser to display the element inline, even if `@Core.ContentDisposition.Filename` is specified, by setting to `inline` (see the fifth example). If omitted, the behavior is `@Core.ContentDisposition.Type: 'attachment'` .

↳ *Learn more how to enable stream support in SAP Fiori elements.*

The following examples show these annotations in action:

1. Media data is stored in a database with a fixed media type `image/png` :

```cds
entity Books { //...
  image : LargeBinary @Core.MediaType: 'image/png';
}
```

2. Media data is stored in a database with a *variable* media type:

```cds
entity Books { //...
  image : LargeBinary @Core.MediaType: imageType;
  imageType : String  @Core.IsMediaType;
}
```

3. Media data is stored in an external repository:

```cds
entity Books { //...
  imageUrl  : String @Core.IsURL @Core.MediaType: imageType;
  imageType : String @Core.IsMediaType;
}
```

4. Content disposition data is stored in a database with a *variable* disposition:

```cds
entity Authors { //...
  image : LargeBinary @Core.MediaType: imageType @Core.ContentDisposition.Fil
  fileName : String;
}
```

5. The image shall have the suggested file name but be displayed inline nevertheless:

```cds
entity Authors { //...
  image : LargeBinary @Core.MediaType: imageType @Core.ContentDisposition.Fil
  fileName : String;
}
```

↳ *Learn more about the syntax of annotations.*

WARNING

> In case you rename the properties holding the media type or content disposition information in a projection, you need to update the annotation's value as well.

Reading Media Resources

Read media data using `GET` requests of the form `/Entity(<ID>)/mediaProperty` :

```cds
GET ../Books(201)/image
> Content-Type: application/octet-stream
```

> The response's `Content-Type` header is typically `application/octet-stream` .

> Although allowed by RFC 2231 , Node.js does not support line breaks in HTTP headers. Hence, make sure you remove any line breaks from your `@Core.IsMediaType` content.

Read media data with `@Core.ContentDisposition.Filename` in the model:

```cds
GET ../Authors(201)/image
> Content-Disposition: 'attachment; filename="foo.jpg"'
```

> The media data is streamed automatically.

↳ *Learn more about returning a custom streaming object (Node.js - beta).*

Creating a Media Resource

As a first step, create an entity without media data using a POST request to the entity. After creating the entity, you can insert a media property using the PUT method. The MIME type is passed in the `Content-Type` header. Here are some sample requests:

```cds
POST ../Books
Content-Type: application/json
{ <JSON> }
```

```cds
PUT ../Books(201)/image
Content-Type: image/png
<MEDIA>
```

> The media data is streamed automatically.

Updating Media Resources

The media data for an entity can be updated using the PUT method:

```cds
PUT ../Books(201)/image
Content-Type: image/png
<MEDIA>
```

> The media data is streamed automatically.

Deleting Media Resources

One option is to delete the complete entity, including all media data:

```http
DELETE ../Books(201)
```

Alternatively, you can delete a media data element individually:

```http
DELETE ../Books(201)/image
```

Using External Resources

The following are requests and responses for the entity containing redirected media data from the third example, "Media data is stored in an external repository".

> This format is used by OData-Version: 4.0. To be changed in OData-Version: 4.01.

```cds
GET: ../Books(201)
>{ ...
    image@odata.mediaReadLink: "http://other-server/image.jpeg",
    image@odata.mediaContentType: "image/jpeg",
    imageType: "image/jpeg"
}
```

Conventions & Limitations

General Conventions

- Binary data in payloads must be a Base64 encoded string.
- Binary data in URLs must have the format `binary'<url-safe base64 encoded>'`. For example:

```http
GET $filter=ID eq binary'Q0FQIE5vZGUuanM='
```

Node.js Runtime Conventions and Limitations

- The usage of binary data in some advanced constructs like the `$apply` query option and `/any()` might be limited.
- On SQLite, binary strings are stored as plain strings, whereas a buffer is stored as binary data. As a result, if in a CDS query, a binary string is used to query data stored as binary, this wouldn't work.
- Please note, that SQLite doesn't support streaming. That means, that LargeBinary fields are read as a whole (not in chunks) and stored in memory, which can impact performance.
- SAP HANA Database Client for Node.js (HDB) and SAP HANA Client for Node.js (`@sap/hana-client`) packages handle binary data differently. For example, HDB automatically converts binary strings into binary data, whereas SAP HANA Client doesn't.
- In the Node.js Runtime, all binary strings are converted into binary data according to SAP HANA property types. To disable this default behavior, you can set the environment variable `cds.hana.base64_to_buffer: false` ☼.

Best Practices

Single-Purposed Services

— Best Practice

We strongly recommend designing your services for single use cases. Services in CAP are cheap, so there's no need to save on them.

DON'T: Single Services Exposing All Entities 1:1

The anti-pattern to that are single services exposing all underlying entities in your app in a 1:1 fashion. While that may save you some thoughts in the beginning, it's likely that it will result in lots of headaches in the long run:

- They open huge entry doors to your clients with only few restrictions
- Individual use-cases aren't reflected in your API design
- You have to add numerous checks on a per-request basis...
- Which have to reflect on the actual use cases in complex and expensive evaluations

DO: One Service Per Use Case

For example, let's assume that we have a domain model defining *Books* and *Authors* more or less as above, and then we add *Orders*. We could define the following services:

```cds
using { my.domain as my } from './db/schema';
```

```cds
/** Serves end users browsing books and place orders */
service CatalogService {
  @readonly entity Books as select from my.Books {
    ID, title, author.name as author
  };
  @requires: 'authenticated-user'
  @insertonly entity Orders as projection on my.Orders;
}
```

```cds
/** Serves registered users managing their account and their orders */
@requires: 'authenticated-user'
service UsersService {
  @restrict: [{ grant: 'READ', where: 'buyer = $user' }] // limit to own ones
  @readonly entity Orders as projection on my.Orders;
  action cancelOrder ( ID:Orders.ID, reason:String );
}
```

```cds
/** Serves administrators managing everything */
@requires: 'authenticated-user'
```

```
service AdminService {
  entity Books   as projection on my.Books;
  entity Authors as projection on my.Authors;
  entity Orders  as projection on my.Orders;
}
```

These services serve different use cases and are tailored for each. Note, for example, that we intentionally don't expose the `Authors` entity to end users.

Late-Cut Microservices — *Best Practice*

Compared to Microservices, CAP services are 'Nano'. As shown in the previous sections, you should design your application as a set of loosely coupled, single-purposed services, which can all be served embedded in a single-server process at first (that is, a monolith).

Yet, given such loosely coupled services, and enabled by CAP's uniform way to define and consume services, you can decide later on to separate, deploy, and run your services as separate microservices, even without changing your models or code.

This flexibility allows you to, again, focus on solving your domain problem first, and avoid the efforts and costs of premature microservice design and DevOps overhead, at least in the early phases of development.