# Using SAP HANA Cloud for Production

**This guide is available for Node.js and Java.**

Use the toggle in the title bar or press `v` to switch.

Table of Contents


- Setup & Configuration
- Running cds build
  - Generated HDI Artifacts
  - Custom HDI Artifacts
- Deploying to SAP HANA
  - Prepare for Production
  - Using cds deploy for Ad-Hoc Deployments
  - Using cf deploy or cf push
- Native SAP HANA Features
  - Vector Embeddings
  - Geospatial Functions
  - Spatial Grid Generators
  - Functions Without Arguments
  - Regex Functions
- HDI Schema Evolution
  - Schema Evolution and Multitenancy/Extensibility

    - Native Database Clauses
    - Advanced Options
- Caveats
    - CSV Data Gets Overridden
    - Undeploying Artifacts
    - SAP HANA Cloud System Limits


SAP HANA Cloud is supported as the CAP standard database and recommended for productive use with full support for schema evolution and multitenancy.

> **WARNING**
>
> CAP isn't validated with other variants of SAP HANA, like "SAP HANA Database as a Service" or "SAP HANA (on premise)".

---

## Setup & Configuration

Run this to use SAP HANA Cloud for production:

```sh
npm add @cap-js/hana
```

▶ *Using other SAP HANA drivers...*

> **Prefer `cds add`**
>
> ... as documented in the deployment guide, which also does the equivalent of `npm add @cap-js/hana` but in addition cares for updating `mta.yaml` and other deployment resources.

## Running `cds build`

Deployment to SAP HANA is done via the SAP HANA Deployment Infrastructure (HDI) which in turn requires running `cds build` to generate all the deployable HDI artifacts. For example, run this in cap/samples/bookshop :

```sh
cds build --for hana
```

Which should display this log output:

```log
[cds] - done > wrote output to:
  gen/db/init.js
  gen/db/package.json
  gen/db/src/gen/.hdiconfig
  gen/db/src/gen/.hdinamespace
  gen/db/src/gen/AdminService.Authors.hdbview
  gen/db/src/gen/AdminService.Books.hdbview
  gen/db/src/gen/AdminService.Books_texts.hdbview
  gen/db/src/gen/AdminService.Currencies.hdbview
  gen/db/src/gen/AdminService.Currencies_texts.hdbview
  gen/db/src/gen/AdminService.Genres.hdbview
  gen/db/src/gen/AdminService.Genres_texts.hdbview
  gen/db/src/gen/CatalogService.Books.hdbview
  gen/db/src/gen/CatalogService.Books_texts.hdbview
  gen/db/src/gen/CatalogService.Currencies.hdbview
  gen/db/src/gen/CatalogService.Currencies_texts.hdbview
  gen/db/src/gen/CatalogService.Genres.hdbview
  gen/db/src/gen/CatalogService.Genres_texts.hdbview
  gen/db/src/gen/CatalogService.ListOfBooks.hdbview
  gen/db/src/gen/data/sap.capire.bookshop-Authors.csv
  gen/db/src/gen/data/sap.capire.bookshop-Authors.hdbtabledata
  gen/db/src/gen/data/sap.capire.bookshop-Books.csv
  gen/db/src/gen/data/sap.capire.bookshop-Books.hdbtabledata
  gen/db/src/gen/data/sap.capire.bookshop-Books.texts.csv
  gen/db/src/gen/data/sap.capire.bookshop-Books.texts.hdbtabledata
  gen/db/src/gen/data/sap.capire.bookshop-Genres.csv
  gen/db/src/gen/data/sap.capire.bookshop-Genres.hdbtabledata
  gen/db/src/gen/localized.AdminService.Authors.hdbview
  gen/db/src/gen/localized.AdminService.Books.hdbview
  gen/db/src/gen/localized.AdminService.Currencies.hdbview
  gen/db/src/gen/localized.AdminService.Genres.hdbview
```

```
gen/db/src/gen/localized.CatalogService.Currencies.hdbview
gen/db/src/gen/localized.CatalogService.Genres.hdbview
gen/db/src/gen/localized.CatalogService.ListOfBooks.hdbview
gen/db/src/gen/localized.sap.capire.bookshop.Authors.hdbview
gen/db/src/gen/localized.sap.capire.bookshop.Books.hdbview
gen/db/src/gen/localized.sap.capire.bookshop.Genres.hdbview
gen/db/src/gen/localized.sap.common.Currencies.hdbview
gen/db/src/gen/sap.capire.bookshop.Authors.hdbtable
gen/db/src/gen/sap.capire.bookshop.Books.hdbtable
gen/db/src/gen/sap.capire.bookshop.Books_author.hdbconstraint
gen/db/src/gen/sap.capire.bookshop.Books_currency.hdbconstraint
gen/db/src/gen/sap.capire.bookshop.Books_foo.hdbconstraint
gen/db/src/gen/sap.capire.bookshop.Books_genre.hdbconstraint
gen/db/src/gen/sap.capire.bookshop.Books_texts.hdbtable
gen/db/src/gen/sap.capire.bookshop.Genres.hdbtable
gen/db/src/gen/sap.capire.bookshop.Genres_parent.hdbconstraint
gen/db/src/gen/sap.capire.bookshop.Genres_texts.hdbtable
gen/db/src/gen/sap.common.Currencies.hdbtable
gen/db/src/gen/sap.common.Currencies_texts.hdbtable
```

## Generated HDI Artifacts

As we see from the log output `cds build` generates these deployment artifacts as expected by HDI , based on CDS models and .csv files provided in your projects:

- `.hdbtable` files for entities
- `.hdbview` files for views / projections
- `.hdbconstraint` files for database constraints
- `.hdbtabledata` files for CSV content
- a few technical files required by HDI, such as `.hdinamespace` and `.hdiconfig`

## Custom HDI Artifacts

In addition to the generated HDI artifacts, you can add custom ones by adding according files to folder `db/src` . For example, let's add an index for Books titles...

1. Add a file `db/src/sap.capire.bookshop.Books.hdbindex` and fill it with this content:

```sql
INDEX sap_capire_bookshop_Books_title_index
ON sap_capire_bookshop_Books (title)
```

2. Run cds build again → this time you should see this additional line in the log output:

```log
[cds] - done > wrote output to:
  [...]
  gen/db/src/sap.capire.bookshop.Books.hdbindex
```

↳ *Learn more about HDI Design-Time Resources and Build Plug-ins*

---

## Deploying to SAP HANA

There are two ways to include SAP HANA in your setup: Use SAP HANA in a hybrid mode, meaning running your services locally and connecting to your database in the cloud, or running your whole application on SAP Business Technology Platform. This is possible either in trial accounts or in productive accounts.

To make the following configuration steps work, we assume that you've provisioned, set up, and started, for example, your SAP HANA Cloud instance in the trial environment . If you need to prepare your SAP HANA first, see How to Get an SAP HANA Cloud Instance for SAP Business Technology Platform, Cloud Foundry environment to learn about your options.

### Prepare for Production

To prepare the project, execute:

```sh
cds add hana --for hybrid
```

This configures deployment for SAP HANA to use the *hdbtable* and *hdbview* formats. The configuration is added to a `[hybrid]` profile in your *package.json*.

> **The profile `hybrid` relates to the hybrid testing scenario**

the Deploy to Cloud Foundry guide.

No further configuration is necessary for Node.js. For Java, see the Use SAP HANA as the Database for a CAP Java Application tutorial for the rest of the configuration.

## Using `cds deploy` for Ad-Hoc Deployments

`cds deploy` lets you deploy *just the database parts* of the project to an SAP HANA instance. The server application (the Node.js or Java part) still runs locally and connects to the remote database instance, allowing for fast development roundtrips.

Make sure that you're logged in to Cloud Foundry with the correct target, that is, org and space. Then in the project root folder, just execute:

```sh
cds deploy --to hana
```

> To connect to your SAP HANA Cloud instance use `cds watch --profile hybrid`.

Behind the scenes, `cds deploy` does the following:

- Compiles the CDS model to SAP HANA files (usually in *gen/db*, or *db/src/gen*)
- Generates *.hdbtabledata* files for the CSV files in the project. If a *.hdbtabledata* file is already present next to the CSV files, no new file is generated.
- Creates a Cloud Foundry service of type `hdi-shared`, which creates an HDI container. Also, you can explicitly specify the name like so: `cds deploy --to hana:<myService>`.
- Starts `@sap/hdi-deploy` locally. If you need a tunnel to access the database, you can specify its address with `--tunnel-address <host:port>`.
- Stores the binding information with profile `hybrid` in the *.cdsrc-private.json* file of your project. You can use a different profile with parameter `--for`. With this information, `cds watch` / `run` can fetch the SAP HANA credentials at runtime, so that the server can connect to it.

Specify `--profile` when running `cds deploy` as follows:

```sh
cds deploy --to hana --profile hybrid
```

Based on these profile settings, `cds deploy` executes `cds build` and also resolves additionally binding information. If a corresponding binding exists, its service name and service key are used. The development profile is used by default.

↪ *Learn more about hybrid testing using service bindings to Cloud services.*

If you run into issues, see the Troubleshooting guide.

### Deploy Parameters

When using the option `--to hana`, you can specify the service name and logon information in several ways.

```
cds deploy --to hana
```

In this case the service name and service key either come from the environment variable `VCAP_SERVICES` or are defaulted from the project name, for example, `myproject-db` with `myproject-db-key`. Service instances and key either exist and will be used, or otherwise they're created.

```
cds deploy --to hana:myservice
```

This overwrites any information coming from environment variables. The service name `myservice` is used and the current Cloud Foundry client logon information is taken to connect to the system.

```
cds deploy --vcap-file someEnvFile.json
```

This takes the logon information and the service name from the `someEnvFile.json` file and overwrite any environment variable that is already set.

```
cds deploy --to hana:myservice --vcap-file someEnvFile.json
```

This is equivalent to `cds deploy --to hana:myservice` and ignores information coming from `--vcap-file`. A warning is printed after deploying.

## Using `cf deploy` or `cf push`

See the Deploying to Cloud Foundry guide for information about how to deploy the complete application to SAP Business Technology Platform, including a dedicated deployer application for the SAP HANA database.

## Native SAP HANA Features

The HANA Service provides dedicated support for native SAP HANA features as follows.

### Vector Embeddings

Vector embeddings are numerical representations that capture important features and semantics of unstructured data - such as text, images, or audio. This representation makes vector embeddings of similar data have high similarity and low distance to each other. These properties of vector embeddings facilitate tasks like similarity search, anomaly detection, recommendations and Retrieval Augmented Generation (RAG). Vector embeddings from a vector datastore such as the SAP HANA Cloud Vector Engine can help get better generative AI (GenAI) results. This is achieved when the embeddings are used as context to the large language models (LLMs) prompts.

Typically vector embeddings are computed using an **embedding model**. The embedding model is specifically designed to capture important features and semantics of a specific type of data, it also determines the dimensionality of the vector embedding space. Unified consumption of embedding models and LLMs across different vendors and open source models is provided via the SAP Generative AI Hub .

In CAP, vector embeddings are stored in elements of type cds.Vector:

```cds
entity Books : cuid {
  title       : String(111);
  description : LargeString;
  embedding   : Vector(1536); // vector space w/ 1536 dimensions
}
```

At runtime, you can compute the similarity and distance of vectors in the SAP HANA vector store using the *cosineSimilarity* and *l2Distance* (Euclidean distance) functions in queries:

Node.js Java

```js
let embedding; // vector embedding as string '[0.3,0.7,0.1,...]';

let similarBooks = await SELECT.from('Books')
  .where`cosine_similarity(embedding, to_real_vector(${embedding})) > 0.9`
```

## Geospatial Functions

CDS supports the special syntax for SAP HANA geospatial functions:

```cds
entity Geo as select from Foo {
  geoColumn.ST_Area() as area : Decimal,
  new ST_Point(2.25, 3.41).ST_X() as x : Decimal
};
```

↳ *Learn more in the SAP HANA Spatial Reference .*

## Spatial Grid Generators

SAP HANA Spatial has some built-in grid generator table functions . To use them in a CDS model, first define corresponding facade entities in CDS.

Example for function *ST_SquareGrid* :

```cds
@cds.persistence.exists
entity ST_SquareGrid(size: Double, geometry: hana.ST_GEOMETRY) {
  geom: hana.ST_GEOMETRY;
  i: Integer;
  j: Integer;
}
```

Then the function can be called, parameters have to be passed by name:

```cds
entity V as select
  from ST_SquareGrid(size: 1.0, geometry: ST_GeomFromWkt('Point(1.5 -2.5)'))
{ geom, i, j };
```

## Functions Without Arguments

SAP HANA allows to omit the parentheses for functions that don't expect arguments. For example:

```
entity Bar as select from Foo {
  ID, current_timestamp
};
```

Some of which are well-known standard functions like `current_timestamp` in the previous example, which can be written without parentheses in CDS models. However, there are many unknown ones, that aren't known to the compiler, for example:

- `current_connection`
- `current_schema`
- `current_transaction_isolation_level`
- `current_utcdate`
- `current_utctime`
- `current_utctimestamp`
- `sysuuid`

To use these in CDS models, you have to add the parentheses so that CDS generic support for using native features can kick in:

cds

```
entity Foo { key ID : UUID; }
entity Bar as select from Foo {
  ID, current_timestamp,
  sysuuid() as sysid
};
```

## Regex Functions

CDS supports SAP HANA Regex functions ( `locate_regexpr` , `occurrences_regexpr` , `replace_regexpr` , and `substring_regexpr` ), and SAP HANA aggregate functions with an additional `order by` clause in the argument list. Example:

sql

```
locate_regexpr(pattern in name from 5)
first_value(name order by price desc)
```

Restriction: `COLLATE` isn't supported.

For other functions, where the syntax isn't supported by the compiler (for example, `xmltable(...)` ), a native *.hdbview* can be used. See Using Native SAP HANA Artifacts for

## HDI Schema Evolution

CAP supports database schema updates by detecting changes to the CDS model when executing the CDS build. If the underlying database offers built-in schema migration techniques, compatible changes can be applied to the database without any data loss or the need for additional migration logic. Incompatible changes like deletions are also detected, but require manual resolution, as they would lead to data loss.

| Change | Detected Automatically | Applied Automatically |
|---|---|---|
| Adding fields | **Yes** | **Yes** |
| Deleting fields | **Yes** | No |
| Renaming fields | n/a [1] | No |
| Changing datatype of fields | **Yes** | No |
| Changing type parameters | **Yes** | **Yes** |
| Changing associations/compositions | **Yes** | No [2] |
| Renaming associations/compositions | n/a [1] | No |
| Renaming entities | n/a | No |

> [1] Rename field or association operations aren't detected as such. Instead, corresponding ADD and DROP statements are rendered requiring manual resolution activities.
>
> [2] Changing targets may lead to renamed foreign keys. Possibly hard to detect data integrity issues due to non-matching foreign key values if target key names remain the same (for example "ID").

**No support for incompatible schema changes**

Currently there's no framework support for incompatible schema changes that require scripted data migration steps (like changing field constraints NULL > NOT NULL). However, the CDS build does detect those changes and renders them as non-executable statements,

changes in productive environments.

## Schema Evolution and Multitenancy/Extensibility

There's full support for schema evolution when the *cds-mtxs* library is used for multitenancy handling. It ensures that all schema changes during base-model upgrades are rolled out to the tenant databases.

> **WARNING**
>
> Tenant-specific extensibility using the *cds-mtxs* library isn't supported yet Right now, you can't activate extensions on entities annotated with `@cds.persistence.journal` .

## Schema Updates with SAP HANA

All schema updates in SAP HANA are applied using SAP HANA Deployment Infrastructure (HDI) design-time artifacts, which are auto-generated during CDS build execution.

Schema updates using *.hdbtable* deployments are a challenge for tables with large data volume. Schema changes with *.hdbtable* are applied using temporary table generation to preserve the data. As this could lead to long deployment times, the support for *.hdbmigrationtable* artifact generation has been added. The Migration Table artifact type uses explicit versioning and migration tasks. Modifications of the database table are explicitly specified in the design-time file and carried out on the database table exactly as specified. This saves the cost of an internal table-copy operation. When a new version of an already existing table is deployed, HDI performs the migration steps that haven't been applied.

### Deploy Artifact Transitions as Supported by HDI

| Current format | hdbcds | hdbtable | hdbmigrationtable |
|---|---|---|---|
| hdbcds | | yes | n/a |
| hdbtable | n/a | | yes |
| hdbmigrationtable | n/a | Yes | |

> Direct migration from *.hdbcds* to *.hdbmigrationtable* isn't supported by HDI. A deployment using *.hdbtable* is required up front.
>
> ↳ *Learn more in the **Enhance Project Configuration for SAP HANA Cloud** section.*
>
> During the transition from *.hdbtable* to *.hdbmigrationtable* you have to deploy version=1 of the *.hdbmigrationtable* artifact, which must not include any migration steps.

HDI supports the *hdbcds* → *hdbtable* → *hdbmigrationtable* migration flow without data loss. Even going back from *.hdbmigrationtable* to *.hdbtable* is possible. Keep in mind that you lose the migration history in this case. For all transitions you want to execute in HDI, you need to specify an undeploy allowlist as described in HDI Delta Deployment and Undeploy Allow List in the SAP HANA documentation.

> **Moving From *.hdbcds* To *.hdbtable***
>
> There a migration guide providing you step-by-step instructions for making the switch.
>
> ↳ *Learn more about Moving From .hdbcds To .hdbtable*

### Enabling hdbmigrationtable Generation for Selected Entities During CDS Build

If you're migrating your already deployed scenario to *.hdbmigrationtable* deployment, you've to consider the remarks in Deploy Artifact Transitions as Supported by HDI.

By default, all entities are still compiled to *.hdbtable* and you only selectively choose the entities for which you want to build *.hdbmigrationtable* by annotating them with `@cds.persistence.journal` .

Example:

cds

```
namespace data.model;

  @cds.persistence.journal
  entity LargeBook {
    key id : Integer;
    title : String(100);
    content : LargeString;
  }
```

CDS build generates *.hdbmigrationtable* source files for annotated entities as well as a *last-dev/csn.json* source file representing the CDS model state of the last build.

> These source files have to be checked into the version control system.

including the required schema update statements to accomplish the new target state. There are cases where you have to resolve or refactor the generated statements, like for reducing field lengths. As they can't be executed without data loss (for example, `String(100)` -> `String(50)`), the required migration steps are only added as comments for you to process explicitly.

Example:

```
>>>> Manual resolution required - DROP statements causing data loss are disab
>>>> by default.
>>>> You may either:
>>>>   uncomment statements to allow incompatible changes, or
>>>>   refactor statements, e.g. replace DROP/ADD by single RENAME statement
>>>> After manual resolution delete all lines starting with >>>>>
-- ALTER TABLE my_bookshop_Books DROP (title);
-- ALTER TABLE my_bookshop_Books ADD (title NVARCHAR(50));
```
txt

Changing the type of a field causes CDS build to create a corresponding ALTER TABLE statement. Data type conversion rules are applied by the SAP HANA database as part of the deployment step. This may cause the deployment to fail if the column contents can't be converted to the new format.

Examples:

1. Changing the type of a field from String to Integer may cause tenant updates to fail if existing content can't be converted.
2. Changing the type of a field from Decimal to Integer can succeed, but decimal places are truncated. Conversion fails if the content exceeds the maximum Integer length.

We recommend keeping *.hdbtable* deployment for entities where you expect low data volume. Every *.hdbmigrationtable* artifact becomes part of your versioned source code, creating a new migration version on every model change/build cycle. In turn, each such migration can require manual resolution. You can switch large-volume tables to *.hdbmigrationtable* at any time, keeping in mind that the existing *.hdbtable* design-time artifact needs to be undeployed.

> **TIP**
>
> Sticking to *.hdbtable* for the actual application development phase avoids lots of initial migration versions that would need to be applied to the database schema.

CDS build performs rudimentary checks on generated *.hdmigrationtable* files:

*.hdbmigrationtable* files and the *last-dev/csn.json* model state. For example, the last migration version not matching the table version is such an inconsistency.

- CDS build fails if manual resolution comments starting with `>>>>>` exist in one of the generated *.hdbmigrationtable* files. This ensures that manual resolution is performed before deployment.

## Native Database Clauses

Not all clauses supported by SQL can directly be written in CDL syntax. To use native database clauses also in a CAP CDS model, you can provide arbitrary SQL snippets with the annotations `@sql.prepend` and `@sql.append` . In this section, we're focusing on schema evolution specific details.

Schema evolution requires that any changes are applied by corresponding ALTER statements. See ALTER TABLE statement reference for more information. A new migration version is generated whenever an `@sql.append` or `@sql.prepend` annotation is added, changed, or removed. ALTER statements define the individual changes that create the final database schema. This schema has to match the schema defined by the TABLE statement in the *.hdbmigrationtable* artifact. Please note that the compiler doesn't evaluate or process these SQL snippets. Any snippet is taken as is and inserted into the TABLE statement and the corresponding ALTER statement. The deployment fails in case of syntax errors.

CDS Model:

```cds
@cds.persistence.journal
@sql.append: 'PERSISTENT MEMORY ON'
entity E {
  ...,
  @sql.append: 'FUZZY SEARCH INDEX ON'
  text: String(100);
}
```

Result in hdbmigrationtable file:

```sql
== version=2
COLUMN TABLE E (
  ...,
  text NVARCHAR(100) FUZZY SEARCH INDEX ON
) PERSISTENT MEMORY ON

== migration=2
```

```
ALTER TABLE E ALTER (text NVARCHAR(100) FUZZY SEARCH INDEX ON);
```

It's important to understand that during deployment new migration versions will be applied on the existing database schema. If the resulting schema doesn't match the schema as defined by the TABLE statement, deployment fails and any changes are rolled-back. In consequence, when removing or replacing an existing `@sql.append` annotation, the original ALTER statements need to be undone. As the required statements can't automatically be determined, manual resolution is required. The CDS build generates comments starting with `>>>>` in order to provide some guidance and enforce manual resolution.

Generated file with comments:

txt

```
== migration=3
>>>>> Manual resolution required - insert ALTER statement(s) as described bel
>>>>> After manual resolution delete all lines starting with >>>>>
>>>>> Insert ALTER statement for: annotation @sql.append of artifact E has be
>>>>> Insert ALTER statement for: annotation @sql.append of element E:text ha
```

Manually resolved file:

sql

```
== migration=3
ALTER TABLE E PERSISTENT MEMORY DEFAULT;
ALTER TABLE E ALTER (text NVARCHAR(100) FUZZY SEARCH INDEX OFF);
```

Appending text to an existing annotation is possible without manual resolution. A valid ALTER statement will be generated in this case. For example, appending the `NOT NULL` column constraint to an existing `FUZZY SEARCH INDEX ON` annotation generates the following statement:

sql

```
ALTER TABLE E ALTER (text NVARCHAR(100) FUZZY SEARCH INDEX ON NOT NULL);
```

> **WARNING**
>
> You can use `@sql.append` to partition your table initially, but you can't subsequently change the partitions using schema evolution techniques as altering partitions isn't supported yet.

The following CDS configuration options are supported to manage *.hdbmigrationtable* generation.

> **WARNING**
>
> This hasn't been finalized yet.

```js
{
  "hana" : {
    "journal": {
      "enable-drop": false,
      "change-mode": "alter" // "drop"
    },
    // ...
  }
}
```

The `"enable-drop"` option determines whether incompatible model changes are rendered as is ( `true` ) or manual resolution is required ( `false` ). The default value is `false` .

The `change-mode` option determines whether `ALTER TABLE ... ALTER` ( `"alter"` ) or `ALTER TABLE ... DROP` ( `"drop"` ) statements are rendered for data type related changes. To ensure that any kind of model change can be successfully deployed to the database, you can switch the `"change-mode"` to `"drop"` , keeping in mind that any existing data will be deleted for the corresponding column. See hdbmigrationtable Generation for more details. The default value is `"alter"` .

## Caveats

### CSV Data Gets Overridden

HDI deploys CSV data as *.hdbtabledata* and assumes exclusive ownership of the data. It's overridden with the next application deployment; hence:

> Only use CSV files for *configuration data* that can't be changed by application users.

Yet, if you need to support initial data with user changes, you can use the `include_filter` option that `.hdbtabledata` offers.

## Undeploying Artifacts

As documented in the HDI Deployer docs, an HDI deployment by default never deletes artifacts. So, if you remove an entity or CSV files, the respective tables, and content remain in the database.

By default, `cds add hana` creates an `undeploy.json` like this:

db/undeploy.json

```json
[
  "src/gen/**/*.hdbview",
  "src/gen/**/*.hdbindex",
  "src/gen/**/*.hdbconstraint",
  "src/gen/**/*_drafts.hdbtable",
  "src/gen/**/*.hdbcalculationview"
]
```

If you need to remove deployed CSV files, also add this entry:

db/undeploy.json

```json
[
  [...]
  "src/gen/**/*.hdbtabledata"
]
```

↳ *See this **troubleshooting** entry for more information.*

## SAP HANA Cloud System Limits

All limitations for the SAP HANA Cloud database can be found in the SAP Help Portal.

Previous page
PostgreSQL

Next page
SAP HANA Native

# Common Types and Aspects

*@sap/cds/common*

CDS ships with a prebuilt model `@sap/cds/common` that provides common types and aspects for reuse.

Table of Contents


- Why Use @sap/cds/common?
  - Outcome = Optimized Best Practice
- Common Reuse Aspects
  - Aspect cuid
  - Aspect managed
  - Aspect temporal
- Common Reuse Types
  - Type Country
  - Type Currency
  - Type Language
  - Type Timezone
- Common Code Lists
  - Aspect CodeList
  - Entity Countries
  - Entity Currencies
  - Entity Languages
  - Entity Timezones

    - Minimalistic Design
- Aspects for Localized Data
    - Aspect TextsAspect
    - Type Locale
    - SQL Persistence
- Providing Initial Data
    - Add Translated Texts
    - Using Tools like Excel
    - Using Prebuilt Content Package
- Adapting to Your Needs
    - Adding Detailed Fields as of ISO 3166-1
    - Protecting Certain Entries
    - Using Different Foreign Keys
    - Mapping to SAP S/4HANA or ABAP Table Signatures
- Adding Own Code Lists
    - Defining a New Code List Entity
    - Defining a New Reuse Type
    - Using the New Reuse Type and Code List
- Code Lists with Validity
    - Accommodating Changes
    - Exclude Outdated Entries from Pick Lists (Optional)


---

## Why Use *@sap/cds/common*?

It's recommended that all applications use the common types and aspects provided through *@sap/cds/common* to benefit from these features:

- **Concise** and **comprehensible** models → see also Conceptual Modeling
- **Foster interoperability** between all applications
- **Proven best practices** captured from real applications
- **Streamlined** data models with **minimal entry barriers**

- **Automatic** support for localized code lists and value helps
- **Extensibility** using Aspects
- **Verticalization** through third-party extension packages

For example, usage is as simple as indicated in the following sample:

```cds
using { Country } from '@sap/cds/common';
entity Addresses {
  street  : String;
  town    : String;
  country : Country; //> using reuse type
}
```

### Outcome = Optimized Best Practice

The final outcomes in terms of modeling patterns, persistence structures, and implementations is essentially the same as with native means, if you would have collected design experiences from prior solutions, such as we did.

> **TIP**
>
> All the common reuse features of *@sap/cds/common* are provided only through this ~100 line .cds model. Additional runtime support isn't required. *@sap/cds/common* merely uses basic CDS modeling features as well as generic features like localized data and temporal data (which only need minimal runtime support with minimal overhead).

In effect, the results are **straightforward**, capturing **best practices** we learned from real business applications, with **minimal footprint**, **optimized performance**, and **maximized adaptability** and **extensibility**.

## Common Reuse Aspects

*@sap/cds/common* defines the following aspects for use in your entity definitions. They give you shortcuts, for concise and comprehensible models, interoperability and out-of-the-box runtime features connected to them.

Use *cuid* as a convenient shortcut, to add canonical, universally unique primary keys to your entity definitions. These examples are equivalent:

```cds
entity Foo : cuid {...}
```

```cds
entity Foo {
  key ID : UUID;
  [...]
}
```

> The service provider runtimes automatically fill in UUID-typed keys like these with auto-generated UUIDs.

↳ *Learn more about **canonical keys** and **UUIDs**.*

## Aspect *managed*

Use *managed*, to add four elements to capture *created by/at* and latest *modified by/at* management information for records. The following examples are equivalent-

```cds
entity Foo : managed {...}
```

```cds
entity Foo {
  createdAt  : Timestamp @cds.on.insert : $now;
  createdBy  : User      @cds.on.insert : $user;
  modifiedAt : Timestamp @cds.on.insert : $now  @cds.on.update : $now;
  modifiedBy : User      @cds.on.insert : $user @cds.on.update : $user;
  [...]
}
```

> **TIP**
>
> *modifiedAt* and *modifiedBy* are set whenever the respective row was modified, that means, also during *CREATE* operations.

The annotations *@cds.on.insert/update* are handled in generic service providers so to fill in those fields automatically.

↳ *Learn more about **generic service features**.*

This aspect basically adds two canonical elements, `validFrom` and `validTo` to an entity. It also adds a tag annotation that connects the CDS compiler's and runtime's built-in support for *Temporal Data*. This built-in support covers handling date-effective records and time slices, including time travel. All you've to do is, add the temporal aspect to respective entities as follows:

```cds
entity Contract : temporal {...}
```

↳ *Learn more about **temporal data**.*

---

## Common Reuse Types

*@sap/cds/common* provides predefined easy-to-use types for *Countries*, *Currencies*, and *Languages*. Use these types in all applications to foster interoperability.

### Type *Country*

The reuse type `Country` is defined in *@sap/cds/common* as a simple managed Association to the code list for countries as follows:

```cds
type Country : Association to sap.common.Countries;
```

Here's an example of how you would use that reuse type:

```cds
using { Country } from '@sap/cds/common';
entity Addresses {
  street  : String;
  town    : String;
  country : Country; //> using reuse type
}
```

The code lists define a key element `code`, which results in a foreign key column `country_code` in your SQL table for Addresses. For example:

```
  street NVARCHAR(5000),
  town NVARCHAR(5000),
  country_code NVARCHAR(3) -- foreign key
);
```

↳ *Learn more about **managed associations**.*

### Type *Currency*

The type for an association to Currencies.

```cds
type Currency : Association to sap.common.Currencies;
```

↳ *It's the same as for `Country`.*

### Type *Language*

The type for an association to Languages.

```cds
type Language : Association to sap.common.Languages;
```

↳ *It's the same as for `Country`.*

### Type *Timezone*

The type for an association to Timezones.

```cds
type Timezone : Association to sap.common.Timezones;
```

↳ *It's the same as for `Country`.*

## Common Code Lists

defined as associations to respective code list entities. They act as code list tables for respective elements in your domain model.

> Note: You rarely have to refer to the code lists in consuming models, but always only do so transitively by using the corresponding reuse types as shown previously.

#### Namespace: *sap.common*

The following definitions are within namespace *sap.common* ...

## Aspect *CodeList*

This is the base definition for the three code list entities in *@sap/cds/common*. It can also be used for your own code lists.

```cds
aspect sap.common.CodeList {
  name  : localized String(111);
  descr : localized String(1111);
}
```

↳ *Learn more about **localized** keyword.*

## Entity *Countries*

The code list entity for countries is meant to be used with ISO 3166-1 **two-letter alpha codes** as primary keys. For example, *'GB'* for the United Kingdom. Nevertheless, it's defined as *String(3)* to allow you to fill in three-letter codes, if needed.

```cds
entity sap.common.Countries : CodeList {
  key code : String(3); //> ISO 3166-1 alpha-2 codes (or alpha-3)
}
```

## Entity *Currencies*

The code list entity for currencies is meant to be used with ISO 4217 **three-letter alpha codes** as primary keys, for example, *'USD'* for US Dollar. In addition, it provides an element to hold the minor unit fractions and for common currency symbols.

```
  key code  : String(3); //> ISO 4217 alpha-3 codes
  symbol    : String(5); //> for example, $, €, £, ₪, ...
  minorUnit : Int16;     //> for example, 0 or 2
}
```

## Entity *Languages*

The code list entity for countries is meant to be used with POSIX locales as defined in ISO/IEC 15897 as primary keys. For example, `'en_GB'` for British English.

cds

```
entity sap.common.Languages : CodeList {
  key code : sap.common.Locale; //> for example, en_GB
}
```

↳ *Learn more on **normalized locales**.*

## Entity *Timezones*

The code list entity for time zones is meant to be used with primary keys like *Area/Location*, as defined in the IANA time zone database . Examples are `America/Argentina/Buenos_Aires` , `Europe/Berlin` , or `Etc/UTC` .

cds

```
entity sap.common.Timezones : CodeList {
  key code : String(100); //> for example, Europe/Berlin
}
```

↳ *Learn more about time zones in Javascript*

↳ *Learn more about time zones in Java*

## SQL Persistence

The following table definition represents the resulting SQL persistence of the `Countries` code list, with the ones for `Currencies` and `Languages` alike:

sql

```
-- the basic code list table
CREATE TABLE sap_common_Countries (
```

```
  descr NVARCHAR(1000),
  code NVARCHAR(3),
  PRIMARY KEY(code)
);
```

### Minimalistic Design

The models for code lists are intentionally minimalistic to keep the entry barriers as low as possible, focusing on the bare minimum of what all applications generally need: a unique code and localizable fields for name and full name or descriptions.

**ISO alpha codes** for languages, countries, and currencies were chosen because they:

1. Are most common (most projects would choose that)
2. Are most efficient (as these codes are also frequently displayed on UIs)
3. Guarantee minimal entry barriers (bringing about 1 above)
4. Guarantee best support (for example, by readable foreign keys)

Assumption is that ~80% of all apps don't need more than what is already covered in this minimalistic model. Yet, in case you need more, you can easily leverage CDS standard features to adapt and extend these base models to your needs as demonstrated in the section Adapting to your needs.

## Aspects for Localized Data

Following are types and aspects mostly used behind the scenes for localized data.
For example given this entity definition:

cds

```
entity Foo {
  key ID : UUID;
  name   : localized String;
  descr  : localized String;
}
```

When unfolding the *localized* fields, we essentially add *.texts* entities in these steps:

```cds
entity Foo.texts : sap.common.TextsAspects { ... }
```

Which in turn unfolds to:

```cds
entity Foo.texts {
  key locale : sap.common.Locale;
}
```

2. Add the primary key of the main entity `Foo` :

```cds
entity Foo.texts {
  key locale : sap.common.Locale;
  key ID : UUID;
}
```

3. Add the localized fields:

```cds
entity Foo.texts {
  key locale : sap.common.Locale;
  key ID : UUID;
  name   : String;
  descr  : String;
}
```

### Namespace: `sap.common`

The following definitions are with namespace `sap.common` ...

## Aspect `TextsAspect`

This aspect is used when generating `.texts` entities for the unfolding of localized elements. It can be extended, which effectively extends all generated `.texts` entities.

```cds
aspect sap.common.TextsAspect {
  key locale: sap.common.Locale;
}
```

↳ *Learn more about **Extending .texts entities**.*

```cds
type sap.common.Locale : String(14) @title: '{i18n>LanguageCode}';
```

The reuse type *sap.common.Locale* is used when generating *.texts* entities for the unfolding of *localized* elements.

↳ *Learn more about **localized data**.*

### SQL Persistence

In addition, the base entity these additional tables and views are generated behind the scenes to efficiently deal with translations:

```sql
-- _texts table for translations
CREATE TABLE Foo_texts (
  ID NVARCHAR(36),
  locale NVARCHAR(14),
  name NVARCHAR(255),
  descr NVARCHAR(1000),
  PRIMARY KEY(ID, locale)
);
```

```sql
-- view to easily read localized texts with automatic fallback
CREATE VIEW localized_Foo AS SELECT
  code,
  COALESCE (localized.name, name) AS name,
  COALESCE (localized.descr, descr) AS descr
FROM Foo (
  LEFT JOIN Foo_texts AS localized
    ON localized.code= code
    AND localized.locale = SESSION_CONTEXT('locale')
)
```

↳ *Learn more about **localized data**.*

## Providing Initial Data

next to your data models.

The following is an example of a *csv* file to provide data for countries:

db/data/sap.common-Countries.csv

```csv
code;name;descr
AU;Australia;Commonwealth of Australia
CA;Canada;Canada
CN;China;People's Republic of China (PRC)
FR;France;French Republic
DE;Germany;Federal Republic of Germany
IN;India;Republic of India
IL;Israel;State of Israel
MM;Myanmar;Republic of the Union of Myanmar
GB;United Kingdom;United Kingdom of Great Britain and Northern Ireland
US;United States;United States of America (USA)
EU;European Union;European Union
```

↳ *Learn more about the database aspects of **Providing Initial Data**.*

## Add Translated Texts

In addition, you can provide translations for the *sap.common.Countries_texts* table as follows:

db/data/sap.common-Countries_texts.csv

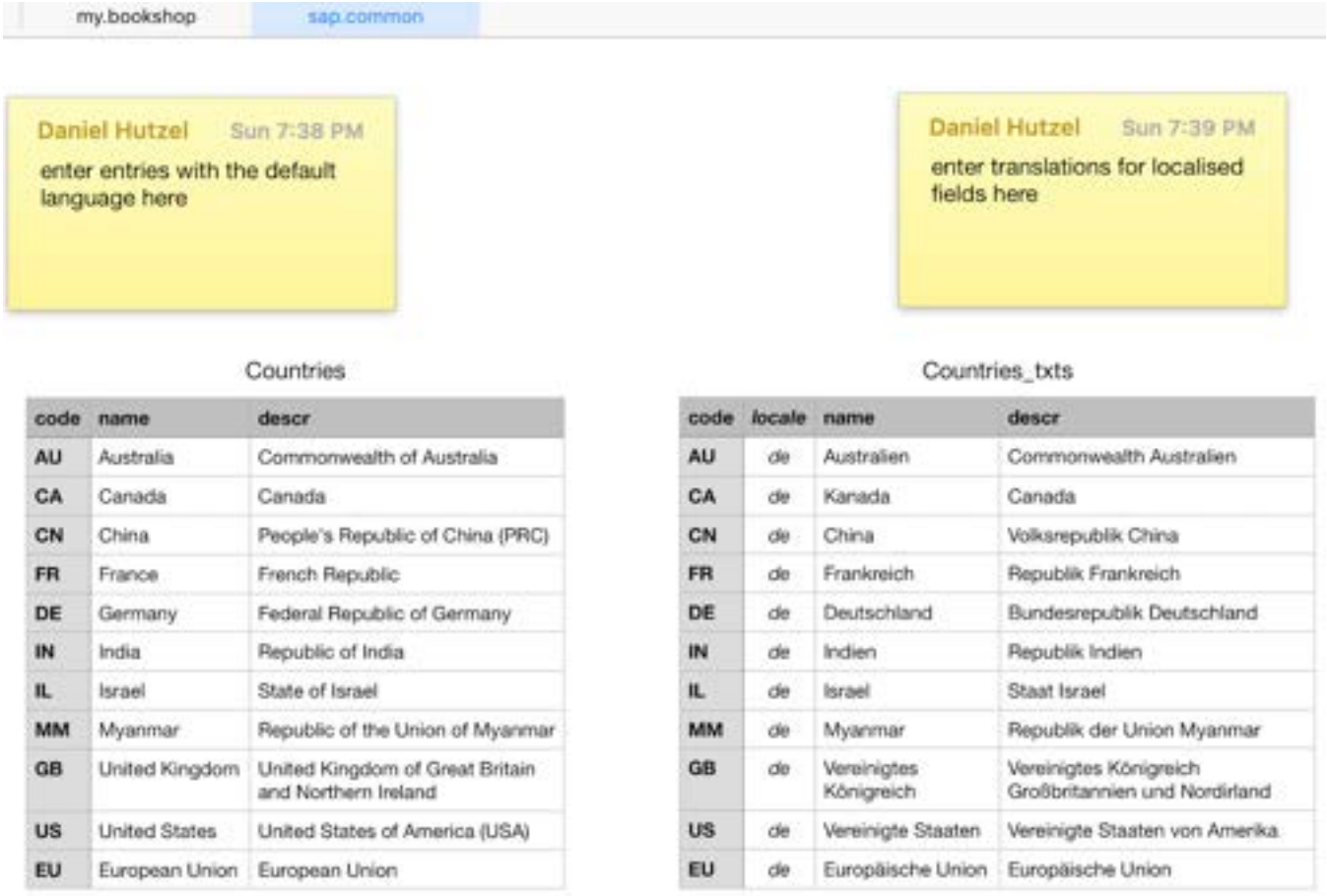
```csv
code;locale;name;descr
AU;de;Australien;Commonwealth Australien
CA;de;Kanada;Canada
CN;de;China;Volksrepublik China
FR;de;Frankreich;Republik Frankreich
DE;de;Deutschland;Bundesrepublik Deutschland
IN;de;Indien;Republik Indien
IL;de;Israel;Staat Israel
MM;de;Myanmar;Republik der Union Myanmar
GB;de;Vereinigtes Königreich;Vereinigtes Königreich Großbritannien und Nordir
```

```
EU;de;Europäische Union;Europäische Union
```

↳ *Learn more about **Localization/i18n**.*

## Using Tools like Excel

You can use Excel or similar tools to maintain these files. For example, the following screenshot shows how we maintained the above two files in Numbers on a Mac:




Countries

| code | name | descr |
|---|---|---|
| AU | Australia | Commonwealth of Australia |
| CA | Canada | Canada |
| CN | China | People's Republic of China (PRC) |
| FR | France | French Republic |
| DE | Germany | Federal Republic of Germany |
| IN | India | Republic of India |
| IL | Israel | State of Israel |
| MM | Myanmar | Republic of the Union of Myanmar |
| GB | United Kingdom | United Kingdom of Great Britain and Northern Ireland |
| US | United States | United States of America (USA) |
| EU | European Union | European Union |

Countries_txts

| code | locale | name | descr |
|---|---|---|---|
| AU | de | Australien | Commonwealth Australien |
| CA | de | Kanada | Canada |
| CN | de | China | Volksrepublik China |
| FR | de | Frankreich | Republik Frankreich |
| DE | de | Deutschland | Bundesrepublik Deutschland |
| IN | de | Indien | Republik Indien |
| IL | de | Israel | Staat Israel |
| MM | de | Myanmar | Republik der Union Myanmar |
| GB | de | Vereinigtes Königreich | Vereinigtes Königreich Großbritannien und Nordirland |
| US | de | Vereinigte Staaten | Vereinigte Staaten von Amerika |
| EU | de | Europäische Union | Europäische Union |

## Using Prebuilt Content Package

Package @sap/cds-common-content provides prebuilt data for the entities `Countries`, `Currencies`, `Languages`, and `Timezones`.

Add it your project:

```sh
npm add @sap/cds-common-content --save
```

```cds
using from '@sap/cds-common-content';
```

↳ *Learn more about integrating reuse packages*

## Adapting to Your Needs

As stated, the predefined definitions are minimalistic by intent. Yet, as *@sap/cds/common* is also just a CDS model, you can apply all the standard features provided by CDS, especially CDS' Aspects to adapt, and extend these definitions to your needs.

Let's look at a few examples of what could be done. You can combine these extensions in an effective model.

> **TIP**
>
> You can do such extensions in the models of your project. You can also collect your extensions into reuse packages and share them as common definitions with several consuming projects, similar to *@sap/cds/common* itself.

↳ *Learn more about providing reuse packages.*

### Adding Detailed Fields as of ISO 3166-1

```cds
using { sap.common.Countries } from '@sap/cds/common';
extend Countries {
  numcode : Integer; //> ISO 3166-1 three-digit numeric codes
  alpha3 : String(3); //> ISO 3166-1 three-letter alpha codes
  alpha4 : String(4); //> ISO 3166-3 four-letter alpha codes
  independent : Boolean;
  status : String(111);
  statusRemark : String(1111);
  remarkPart3 : String(1111);
}
```

> Value lists in SAP Fiori automatically search in the new text fields as well.

Some application logic might have to be hard-coded against certain entries in code lists. Therefore, these entries have to be protected against changes and removal. For example, let's assume a code list for payment methods defined as follows:

```cds
entity PaymentMethods : sap.common.CodeList {
  code : String(11);
}
```

Let's further assume the entires with code *`Main`* and *`Travel`* are required by implementations and hence must not be changed or removed. Have a look at a couple of solutions.

### Programmatic Solution

A fallback, and at the same time, the most open, and most flexible approach, is to use a custom handler to assert that. For example, in Node.js:

```js
srv.on ('DELETE', 'PaymentMethods', req=>{
  const entry = req.query.DELETE.where[2].val
  if (['Main','Travel'].includes(entry))
    return req.reject(403, 'these entries must not be deleted')
})
```

## Using Different Foreign Keys

Let's assume you prefer to have references to the latest code list entries without adjusting foreign keys. This can be achieved by adding and using numeric ISO codes for foreign keys instead of the alpha codes.

your-common.2.cds

```cds
namespace your.common;
using { sap.common.Countries } from '@sap/cds/common';

// Extend Countries code list with fields for numeric codes
extend Countries {
  numcode : Integer; //> ISO 3166-1 three-digit numeric codes
}
```

```
type Country : Association to Countries { numcode };
```

You can use your own definition of *`Country`* instead of the one from *@sap/cds/common* in your models as follows:

```cds
using { your.common.Country } from './your-common.2';

entity Addresses {
  //...
  country : Country;
}
```

### Mapping to SAP S/4HANA or ABAP Table Signatures

```cds
using { sap.common.Countries } from '@sap/cds/common';
entity Countries4GFN as projection on Countries {
  code as CountryCodeAlpha2,
  name as CountryShortName,
  // ...
}
entity Countries4ABAP as projection on Countries {
  code as LAND,
  // ...
}
```

These views are updatable on SAP HANA and many other databases. You can also use CDS to expose them through corresponding OData services in order to ease integration with SAP S/4HANA or older ABAP backends.

## Adding Own Code Lists

As another example of adaptations, let's add support for subdivisions, that means regions, as of ISO 3166-2 to countries.

your-common.4.1.cds

```cds
using sap from '@sap/cds/common';

// new code list for regions
entity Regions : sap.common.CodeList {
  key code : String(5); // ISO 3166-2 alpha5 codes, like DE-BW
  country  : Association to sap.common.Countries;
}

// bi-directionally associate Regions with Countries
extend sap.common.Countries {
  regions : Composition of many Regions on regions.country = $self;
}
```

*`Regions`* is a new, custom-defined code list entity defined in the same way as the predefined ones in *@sap/cds/common*. In particular, it inherits all elements and annotations from the base definition *`sap.common.CodeList`*. For example, the *`@cds.autoexpose`* annotation, which provides that *`Regions`* is auto-exposed in any OData service that has exposed entities with associations to it. The localization of the predefined elements *`name`* and *`descr`* is also inherited.

↳ *Learn in our sample how an own code list can be used to localize `enum` values.*

## Defining a New Reuse Type

Following the pattern for codes in *@sap/cds/common* a bit more, you can also define a reuse type for regions as a managed association:

your-common.4.2.cds

```cds
using { Regions } from './your-common.4.1'; /*>skip<*/
// Define an own reuse type referring to Regions
type Region : Association to Regions;
```

## Using the New Reuse Type and Code List

reuse types. These elements receive the same support from built-in generic features. For example:

```cds
using { Country, Region } from './your-common.4.2';
entity Addresses {
  street  : String;
  town    : String;
  country : Country; //> pre-defined reuse type
  region  : Region; //> your custom reuse type
}
```

## Code Lists with Validity

Even ISO codes may change over time and you may have to react to that in your applications. For example, when Burma was renamed to Myanmar in 1989. Let's investigate strategies on how that can be updated in our code lists.

### Accommodating Changes

The renaming from Burma to Myanmar in 1989, was reflected in ISO 3166 as follows (*the alpha-4 codes as specified in ISO 3166-3 signify entries officially deleted from ISO 3166-1 code lists*):

| Name | Alpha-2 | Alpha-3 | Alpha-4 | Numeric |
|---|---|---|---|---|
| Burma | BU | BUR | BUMM | 104 |
| Myanmar | MM | MMR | | 104 |

By default, and with the given default definitions in *@sap/cds/common*, this would have been reflected as a new entry for Myanmar and you'd have the following choices on what to do with the existing records in your data:

- **(a)** Adjust foreign keys for records so that it always reflects the current state.

the time they were created or valid.

## Exclude Outdated Entries from Pick Lists (Optional)

Although outdated entries like the one for Burma have to remain in the code lists as targets for references from historic records in other entities, you would certainly want to exclude it from all pick lists used in UIs when entering new data. This is how you could achieve that:

### 1. Extend the Common Code List Entity

```cds
using { sap.common.Countries } from '@sap/cds/common';
extend Countries with { validTo: Date default '9999-12-31'; }
```

### 2. Fill Validity Boundaries in Code Lists:

| code | name | validTo |
|---|---|---|
| BU | Burma | 1989-06-18 |
| MM | Myanmar | 9999-12-31 |

### 3. Model Pick List Entity

Add the following line to your service definition:

```cds
entity CountriesPickList as projection on sap.common.Countries where validTo
```

Basically, the entity `Countries` serves all standard requests, and the new entity `CountriesPickList` is built for the value help only. This entity is a projection that gives you only those records that are valid right now.

### 4. Include Pick List with Validity on the UI

This snippet equips UI fields for a `countries` association with a value help from the `CountriesPickList` entity.

```
  countries @(
    Common: {
      Text: country.name , // TextArrangement: #TextOnly,
      ValueList: {
        Label: 'Country Value Help',
        CollectionPath: 'CountriesPickList',
        Parameters: [
          { $Type: 'Common.ValueListParameterInOut',
            LocalDataProperty: country_code,
            ValueListProperty: 'code'
          },
          { $Type: 'Common.ValueListParameterDisplayOnly',
            ValueListProperty: 'name'
          }
      ]
      }
    },
  );
}
```

Edit this page

Last updated: 7/16/24, 2:35 PM

Previous page
Built-in Types

Next page
Aspects vs. Inheritance

# Query Language (CQL)

CDS Query Language (CQL) is based on standard SQL, which it enhances by...

Table of Contents


- Postfix Projections
  - Nested Expands beta
  - Nested Inlines beta
- Smart * Selector
  - Excluding Clause
  - In Nested Expands beta
  - In Nested Inlines beta
- Path Expressions
  - Path Expressions in from Clauses
  - Path Expressions in All Other Clauses
  - With Infix Filters
  - Exists Predicate
- Casts in CDL
- Association Definitions
  - Query-Local Mixins
  - In the select list

## Postfix Projections

CQL allows to put projections, that means, the *SELECT* clause, behind the *FROM* clause enclosed in curly braces. For example, the following are equivalent:

```sql
SELECT name, address.street from Authors
```

```sql
SELECT from Authors { name, address.street }
```

## Nested Expands beta

Postfix projections can be appended to any column referring to a struct element or an association and hence be nested. This allows **expand** results along associations and hence read deeply structured documents:

```sql
SELECT from Authors {
   name, address { street, town { name, country }}
};
```

This actually executes three correlated queries to authors, addresses, and towns and returns a structured result set like that:

```js
results = [
  {
    name: 'Victor Hugo',
    address: {
      street: '6 Place des Vosges', town: {
        name: 'Paris',
        country: 'France'
      }
    }
  }, {
    name: 'Emily Brontë', …
  }, …
]
```

> This is rather a feature tailored to NoSQL databases and has no equivalent in standard SQL as it requires structured result sets. Some SQL vendors allow things like that with non-

> **WARNING**
>
> Nested Expands following *to-many* associations are not supported.

### Alias

As the name of the struct element or association preceding the postfix projection appears in the result set, an alias can be provided for it:

```sql
SELECT from Authors {
   name, address as residence { street, town as city { name, country }}
};
```

The result set now is:

```js
results = [
  {
    name: 'Victor Hugo',
    residence: {
      street: '6 Place des Vosges', city: {
        name: 'Paris',
        country: 'France'
      }
    }
  }, …
]
```

### Expressions

Nested Expands can contain expressions. In addition, it's possible to define new structures that aren't present in the data source. In this case an alias is mandatory and is placed *behind* the *{…}* :

```sql
SELECT from Books {
   title,
   author { name, dateOfDeath - dateOfBirth as age },
   { stock as number, stock * price as value } as stock
};
```

```js
results = [
  {
    title: 'Wuthering Heights',
    author: {
      name: 'Emily Brontë',
      age: 30
    },
    stock: {
      number: 12,
      value: 133.32
    }
  }, …
]
```

## Nested Inlines beta

Put a `"."` before the opening brace to **inline** the target elements and avoid writing lengthy lists of paths to read several elements from the same target. For example:

```sql
SELECT from Authors {
   name, address.{ street, town.{ name, country }}
};
```

… is equivalent to:

```sql
SELECT from Authors {
  name,
  address.street,
  address.town.name,
  address.town.country
};
```

Nested Inlines can contain expressions:

```sql
SELECT from Books {
   title,
   author.{
     name, dateOfDeath - dateOfBirth as author_age,
```

```
    }
};
```

The previous example is equivalent to the following:

```sql
SELECT from Books {
   title,
   author.name,
   author.dateOfDeath - author.dateOfBirth as author_age,
   concat(author.address.town.name, '/', author.address.town.country) as auth
};
```

## Smart * Selector

Within postfix projections, the `*` operator queries are handled slightly different than in plain SQL select clauses.

**Example:**

```sql
SELECT from Books { *, author.name as author }
```

Queries like in our example, would result in duplicate element effects for *`author`* in SQL. In CQL, explicitly defined columns following an `*` replace equally named columns that have been inferred before.

### Excluding Clause

Use the *`excluding`* clause in combination with *`SELECT *`* to select all elements except for the ones listed in the exclude list.

```sql
SELECT from Books { * } excluding { author }
```

example, assume the following definitions:

```cds
entity Foo { foo : String; bar : String; car : String; }
entity Bar as select from Foo excluding { bar };
entity Boo as select from Foo { foo, car };
```

A *SELECT * from Bar* would result into the same as a query of *Boo* :

```sql
SELECT * from Bar --> { foo, car }
SELECT * from Boo --> { foo, car }
```

Now, assume a consumer of that package extends the definitions as follows:

```cds
extend Foo with { boo : String; }
```

With that, queries on *Bar* and *Boo* would return different results:

```sql
SELECT * from Bar --> { foo, car, boo }
SELECT * from Boo --> { foo, car }
```

## In Nested Expands beta

If the * selector is used following an association, it selects all elements of the association target. For example, the following queries are equivalent:

```sql
SELECT from Books { title, author { * } }
```

```sql
SELECT from Books { title, author { ID, name, dateOfBirth, … } }
```

A * selector following a struct element selects all elements of the structure and thus is equivalent to selecting the struct element itself. The following queries are all equivalent:

```sql
SELECT from Authors { name, struc { * } }
SELECT from Authors { name, struc { elem1, elem2 } }
SELECT from Authors { name, struc }
```

The *excluding* clause can also be used for Nested Expands:

### In Nested Inlines beta

The expansion of `*` in Nested Inlines is analogous. The following queries are equivalent:

```sql
SELECT from Books { title, author.{ * } }
SELECT from Books { title, author.{ ID, name, dateOfBirth, … } }
```

The *excluding* clause can also be used for Nested Inlines:

```sql
SELECT from Books { title, author.{ * } excluding { dateOfDeath, placeOfDeath
```

## Path Expressions

Use path expressions to navigate along associations and/or struct elements in any of the SQL clauses as follows:

In *from* clauses:

```sql
SELECT from Authors[name='Emily Brontë'].books;
SELECT from Books:authors.towns;
```

In *select* clauses:

```sql
SELECT title, author.name from Books;
SELECT *, author.address.town.name from Books;
```

In *where* clauses:

```sql
SELECT from Books where author.name='Emily Brontë'
```

## Path Expressions in *from* Clauses

Path expressions in from clauses allow to fetch only those entries from a target entity, which are associated to a parent entity. They unfold to *SEMI JOINS* in plain SQL queries. For example, the previous mentioned queries would unfold to the following plain SQL counterparts:

```sql
SELECT * from Books WHERE EXISTS (
  SELECT 1 from Authors WHERE Authors.ID = Books.author_ID
    AND Authors.name='Emily Brontë'
);
```

```sql
SELECT * from Towns WHERE EXISTS (
  SELECT 1 from Authors WHERE Authors.town_ID = Towns.ID AND EXISTS (
    SELECT 1 from Books WHERE Books.author_ID = Authors.ID
  )
);
```

## Path Expressions in All Other Clauses

Path expressions in all other clauses are very much like standard SQL's column expressions with table aliases as single prefixes. CQL essentially extends the standard behavior to paths with multiple prefixes, each resolving to a table alias from a corresponding *LEFT OUTER JOIN* . For example, the path expressions in the previous mentioned queries would unfold to the following plain SQL queries:

```sql
-- plain SQL
SELECT Books.title, author.name from Books
LEFT JOIN Authors author ON author.ID = Books.author_ID;
```

```sql
-- plain SQL
SELECT Books.*, author_address_town.name from Books
LEFT JOIN Authors author ON author.ID = Books.author_ID
LEFT JOIN Addresses author_address ON author_address.ID = author.address_ID
LEFT JOIN Towns author_address_town ON author_address_town.ID = author_addres
```

```
SELECT Books.* from Books
LEFT JOIN Authors author ON author.ID = Books.author_ID
WHERE author.name='Emily Brontë'
```

> **TIP**
>
> All column references get qualified → in contrast to plain SQL joins there's no risk of ambiguous or conflicting column names.

## With Infix Filters

Append infix filters to associations in path expressions to narrow the resulting joins. For example:

```sql
SELECT books[genre='Mystery'].title from Authors
 WHERE name='Agatha Christie'
```

... unfolds to:

```sql
SELECT books.title from Authors
LEFT JOIN Books books ON ( books.author_ID = Authors.ID )
  AND ( books.genre = 'Mystery' )  //--> from Infix Filter
WHERE Authors.name='Agatha Christie';
```

If an infix filter effectively reduces the cardinality of a *to-many* association to *one*, make this explicit with:

```sql
SELECT name, books[1: favorite=true].title from Authors
```

## Exists Predicate

Use a filtered path expression to test if any element of the associated collection matches the given filter:

```sql
SELECT FROM Authors {name} WHERE EXISTS books[year = 2000]
```

```sql
SELECT name FROM Authors
WHERE EXISTS (
        SELECT 1 FROM Books
        WHERE Books.author_id = Authors.id
            AND Books.year = 2000
    )
```

Exists predicates can be nested:

```sql
SELECT FROM Authors { name }
    WHERE EXISTS books[year = 2000 and EXISTS pages[wordcount > 1000]]
```

A path with several associations is rewritten as nested exists predicates. The previous query is equivalent to the following query.

```sql
SELECT FROM Authors { name }
    WHERE EXISTS books[year = 2000].pages[wordcount > 1000]
```

> **WARNING**
>
> Paths *inside* the filter are not yet supported.

## Casts in CDL

There are two different constructs commonly called casts. SQL casts and CDL casts. The former produces SQL casts when rendered into SQL, whereas the latter does not:

```sql
SELECT cast (foo+1 as Decimal) as bar from Foo;  -- standard SQL
SELECT from Foo { foo+1 as bar : Decimal };      -- CDL-style
```

↳ *learn more about CDL type definitions*

Use SQL casts when you actually want a cast in SQL. CDL casts are useful for expressions such as `foo+1` as the compiler does not deduce types. For the OData backend, by

EDM(X) files.

> **TIP**
>
> You don't need a CDL cast if you already use a SQL cast. The compiler will extract the type from the SQL cast.

## Association Definitions

### Query-Local Mixins

Use the *mixin...into* clause to logically add unmanaged associations to the source of the query, which you can use and propagate in the query's projection. This is only supported in postfix notation.

```sql
SELECT from Books mixin {
  localized : Association to LocalizedBooks on localized.ID = ID;
} into {
  ID, localized.title
};
```

### In the select list

Define an unmanaged association directly in the select list of the query to add the association to the view's signature. This association cannot be used in the query itself. In contrast to mixins, these association definitions are also possible in projections.

```cds
entity BookReviews as select from Reviews {
  ...,
  subject as bookID,
  book : Association to Books on book.ID = bookID
};
```

list. Elements of the query's data sources are not accessible.

This syntax can also be used to add new unmanaged associations to a projection or view via *extend* :

```cds
extend BookReviews with columns {
  subject as bookID,
  book : Association to Books on book.ID = bookID
};
```

Edit this page

Last updated: 7/16/24, 2:35 PM

Previous page
Schema Notation (CSN)

Next page
Query Notation (CQN)

# Definition Language (CDL)

Find here a reference of all CDS concepts and features in the form of compact examples. The examples are given in ***CDL***, a human-readable syntax for defining models, and ***CQL***, an extension of SQL to write queries.

Table of Contents


- Entity and Type Definitions
  - Entity Definitions – define entity
  - Type Definitions – define type
  - Predefined Types
  - Structured Types
  - Arrayed Types
  - Virtual Elements
  - Literals
  - Delimited Identifiers
  - Calculated Elements
  - Default Values
  - Type References
  - Constraints
  - Enums
- Views and Projections
  - The as select from Variant
  - The as projection on Variant
  - Views with Inferred Signatures
  - Views with Parameters
- Associations & Compositions

    - 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 beta
    - The annotate Directive
    - Extend Array Annotations
- Aspects
    - The extend Directive
    - Named Aspects – define aspect
    - Includes -- : as Shortcut Syntax
    - Looks Like Inheritance
    - Extending Views and Projections
- Services
    - Service Definitions
    - Exposed Entities
    - (Auto-) Redirected Associations
    - Auto-Exposed Entities
    - Custom Actions and Functions
    - Custom-Defined Events
    - Extending Services
- Namespaces
    - The namespace Directive
    - The context Directive
    - Scoped Definitions
    - Fully Qualified Names
- Import Directives

    - Model Resolution
- Comments
    - Single-Line Comments – //
    - Multi-Line Comments – /* */
    - Doc Comments – /** */


Refer also to *The Nature of Models* and the *CSN specification* to complete your understanding of CDS.

---

## Entity and Type Definitions


- Entity Definitions – `define entity`
- Type Definitions – `define type`
- Predefined Types
- Structured Types
- Arrayed Types
- Virtual Elements
- Literals
- Delimited Identifiers
- Calculated elements
- Default Values
- Type References
- Constraints
- Enums


### Entity Definitions – `define entity`

(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 – *define type*

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, structure 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**.*

## Predefined Types

↳ *See list of **Built-in Types***

## Structured Types

You can declare and use custom struct types as follows:

```
  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;
  };
}
```

## 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.

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.

cds

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

Using literals in CDS models is commonly used, for example, to set default values. The literals in the following table show you how to define these values in your CDS source.

| Kind | Example |
| --- | --- |
| Null | `null` |
| Boolean | `true` , `false` |
| Numbers | `11` , `2.4` , or `1.34e10` |
| Strings | `'foo'` or `` `foo` `` or ```` ```foo``` ```` |
| Dates | `date'2016-11-24'` |
| Times | `time'16:11:32Z'` |
| Timestamp | `timestamp'2016-11-24T16:11:32.4209753Z'` |
| DateTime | `'2016-11-24T16:11Z'` |
| Records | `{"foo":<literal>, ...}` |
| Arrays | `[<literal>, ...]` |

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

### String Literals

String literals enclosed in single ticks, for example `'string'` , are limited to a single line. A single tick `'` inside the literal is escaped by doubling it: `'it''s escaped` .

Use string literals enclosed in single or triple **backticks** for multiline strings. 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 ( `\` ). Only for string literals inside triple backticks, indentation is stripped and tagging is possible.

**Examples:**

cds

```
@documentation: ```
    This is a CDS multiline string.
    - The indentation is stripped.
    - \u{0055}nicode escape sequences are possible,
```

```
    \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>
  ```
@escaped: `OK Emoji: \u{1f197}`
entity DocumentedEntity {
  // ...
}
```

## Delimited Identifiers

Delimited identifiers allow you to use any identifier, even containing special characters or using a keyword.

> **WARNING**
>
> Special characters in identifiers or keywords as identifiers should be avoided for best interoperability.

cds

```
entity ![Entity] {
  bar           : ![Keyword];
  ![with space] : Integer;
}
```

> You can escape `]` by `]]`, for example `![L[C]]R]` which will be parsed as `L[C]R`.

## 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

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 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)
);
```

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 or composition, like in this example:

```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 a 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 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 and 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
)
```

```
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,
```

```
};
```

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

## 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;
```

↳ *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 & Compositions

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;
}
```

`address_ID` being added automatically upon activation to a SQL database.

> 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 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 of 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 compostions of one are involved. You must fill in foreign keys manually in a custom handler.

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;
```

```
}
```

### 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.

```
  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 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
};
```

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
- The `annotate` Directive
- Extend Array Annotations

### Annotation Syntax

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.

## Annotation Targets

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

Contexts and services:

```cds
@before [define] (context|service) Foo @inner { ... }
```

Definitions and elements with simple types:

```cds
@before [define] type Foo @inner : String @after;
@before [key] anElement @inner : String @after;
```

Entities, aspects, and other struct types and elements thereof:

```cds
@before [define] (entity|type|aspect|annotation) Foo @inner {
  @before simple @inner : String @after;
```

```
}
```

Enums:

```cds
… status : String @inner enum {
  fulfilled @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:

```
@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 or resolved 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 }
@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.

in our previous example.

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

## Expressions as Annotation Values beta

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.

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.
- If a path cannot 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.

### 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 sometimes is 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 set to `true` .

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 are situations where automatic rewriting doesn't work, resulting in the compiler error `anno-missing-rewrite`. Some of these situations are going to be addressed in upcoming releases.

### 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

Expressions in OData Annotations.

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

You can also directly annotate a single element:

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

Actions, functions, their parameters and *returns* can be annotated:

cds
```
service SomeService {
  entity SomeEntity { key id: Integer } actions
  {
    action boundAction(P: Integer) returns String;
  };
  action unboundAction(P: Integer) returns String;
};

annotate SomeService.unboundAction with @label: 'Action Label' (@label: 'Firs
                                        returns @label: 'Returns a string';
annotate SomeService.SomeEntity with actions {
     @label: 'Action label'
     boundAction(@label: 'firstParameter' P) returns @label: 'Returns a strin
}
```

The *annotate* directive is a variant of the *extend* directive. Actually, *annotate* is just a shortcut with the default mode being switched to *extend* ing existing fields instead of adding new ones.

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
```

```
];
```

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', L
    { 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
- Named Aspects – *define aspect*
- Shortcut Syntax *:*
- Looks Like Inheritance
- Extending Views / Projections


### 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') {
  newField : String;
  extend nestedStructField {
    newField : String;
    extend existingField @title:'Nested Field';
  }
}
extend Bar with @title: 'Bar'; // nothing for elements
```

> **TIP**
>
> Make sure that you prepend the *extend* keyword to nested elements, otherwise this would mean that you want to add a new field with that name:

↳ *Learn more about the **annotate** Directive.*

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:

```
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.

## Named Aspects – *define aspect*

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

cds
```
extend Foo with ManagedObject;
extend Bar with ManagedObject;
```

cds
```
aspect ManagedObject {
  created { at: Timestamp; _by: User; }
}
```

> The *define* keyword is optional, that means *define aspect Foo* is equal to *aspect Foo*.

If you use *extend*, all nested fields in the named aspect are interpreted as being extension fields. If you use *annotate*, the nested fields are interpreted as existing fields and the annotations are copied to the corresponding target elements.

The named extension can be anything, for example, including other *types* or *entities*. 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.

## 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 : ManagedObject, 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 ManagedObject;
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.

## Looks Like Inheritance

The `:` -based syntax option described before looks very much like (multiple) inheritance and in fact has very much the same effects. Yet, as mentioned in the beginning of this section, it isn't based on inheritance but on mixins, which are more powerful and also avoid common problems like the infamous diamond shapes in type derivations.

When combined with persistence mapping there are a few things to note, that goes down to which strategy to choose to map inheritance to, for example, relational models. See *Aspects vs Inheritance* for more details.

## 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 Foo with @title:'Foo' columns {
  foo as moo @woo,
  1 + 1 as two,
```

```
}
```

> **TIP**
>
> Enhancing nested structs isn't supported. Note also that you can use the common `annotate` syntax, to just add/override annotations of a view's elements.

---

# Services

- Service Definitions
- Exposed Entities
- (Auto-) Redirected Associations
- Auto-exposed Targets
- Custom Actions/Functions
- Custom-defined Events
- Extending 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:

```
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 *`view with parameter`* modeled in the previous example, 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
}
```

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;
```

## 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 { /*...*/ }
```

```
}
```

↳ *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 { ... }
```

```
      // 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 by *`[many] $self`* . Use Explicit Binding to control the naming of the binding parameter. Use the keyword *`many`* to indicate that the action or function is bound to a collection of instances rather than to a single one.

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.

## 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, type, or another event. Only the effective signature of the projection is relevant.

```
  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;
}
```

## Namespaces


- The *namespace* Directive
- The *context* Directive
- Scoped Definitions
- Fully Qualified Names


### 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.

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 second model above would compile to this CSN:

```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" }
}}
```

## Import Directives


- The *using* Directive
- Model Resolution


### 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
```

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.

## Comments

- Single-Line Comments
- Multi-Line Comments
- Doc comments

Any text between `//` and the end of the line is ignored:

```cds
entity Employees {
  key ID : Integer;  // a single-line comment
  name : String;
}
```

## Multi-Line Comments – `/* */`

Any text between `/*` and `*/` is ignored:

```cds
entity Employees {
  key ID : Integer;
/*
  a multi-line comment
*/
  name : String;
}
```

unless it is a 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;
}
```

EDM(X), it appears as value for the annotation `@Core.Description` .

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"'
```

> **TIP**
>
> Propagation of doc comments can be stopped via an empty one: `/** */` .

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

CLI package.json JavaScript

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

> **Doc comments are enabled by default 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 .

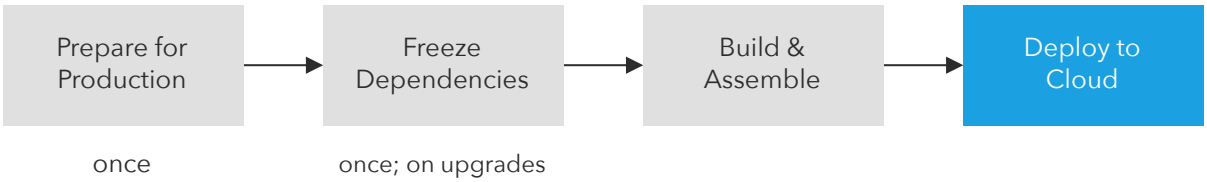
# Deploy to Cloud Foundry

A comprehensive guide on deploying applications built with SAP Cloud Application Programming Model (CAP) to SAP BTP Cloud Foundry environment.

## Table of Contents


- Intro & Overview
- Prerequisites
- Prepare for Production
  - 1. Using SAP HANA Database
  - 2. Using XSUAA-Based Authentication
  - 3. Using MTA-Based Deployment
  - 4. Using App Router as Gateway
  - 5. Optional: Add Multitenancy
  - 6. Freeze Dependencies
- Build & Assemble
  - Build Deployables with cds build
  - Assemble with mbt build
- Deploy to Cloud
  - Inspect Deployed Apps in BTP Cockpit
- Deploy using cf push
  - Prerequisites
  - Add a manifest.yml
  - Build the Project
  - Push the Application

After completing the functional implementation of your CAP application by following the Getting Started or Cookbook guides, you would finally deploy it to the cloud for production. The essential steps are illustrated in the following graphic:




First, you apply these steps manually in an ad-hoc deployment, as described in this guide. Then, after successful deployment, you automate them using CI/CD pipelines.

> **This guide is available for Node.js and Java.**
>
> Use the toggle in the title bar or press `v` to switch.

## Prerequisites

The following sections are based on our *cap/samples/bookshop* project. Download or clone the repository, and exercise the following steps in the *bookshop* subfolder:

```sh
git clone https://github.com/sap-samples/cloud-cap-samples samples
cd samples/bookshop
```

In addition, you need to prepare the following:

### 1. SAP BTP with SAP HANA Cloud Database up and Running

- Access to SAP BTP, for example a trial
- An SAP HANA Cloud database running in your subaccount

- A Cloud Foundry space

> **WARNING**
>
> As starting the SAP HANA database takes several minutes, we recommend doing these steps early on. In trial accounts, you need to start the database **every day**.

### 2. Latest Versions of *@sap/cds-dk*

Likewise, ensure the latest version of *@sap/cds* is installed in your project:

```sh
npm outdated          #> check whether @sap/cds is listed
npm i @sap/cds        #> if necessary
```

### 3. Cloud MTA Build Tool

- Run *mbt* in a terminal to check whether you've installed it.
- If not, install it according to the MTA Build Tool's documentation .
- For macOS/Linux machines best is to install using *npm* :

```sh
npm i -g mbt
```

- For Windows, please also install *GNU Make*

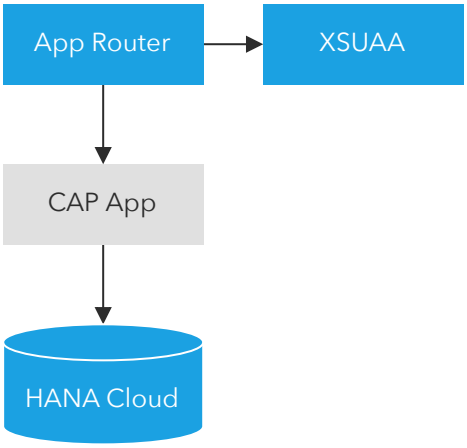
### 4. Cloud Foundry CLI w/ MTA Plugins

- Run *cf -v* in a terminal to check whether you've installed version **8** or higher.
- If not, install or update it according to the Cloud Foundry CLI documentation .
- In addition, ensure to have the MTA plugin for the Cloud Foundry CLI installed.

```sh
cf add-plugin-repo CF-Community https://plugins.cloudfoundry.org
cf install-plugin multiapps
```

## Prepare for Production

If you followed CAP's grow-as-you-go approach so far, you've developed your application with an in-memory database and basic/mock authentication. To prepare for production you need to ensure respective production-grade choices are configured, as illustrated in the following graphic:




> We'll use the `cds add <facets>` CLI command for that, which ensures the required services are configured correctly and corresponding package dependencies are added to your *package.json*.

### 1. Using SAP HANA Database

While we used SQLite as a low-cost stand-in during development, we're going to use a managed SAP HANA database for production:

```sh
cds add hana --for production
```

↳ *Learn more about using SAP HANA for production.*

### 2. Using XSUAA-Based Authentication

Configure your app for XSUAA-based authentication:

```sh
cds add xsuaa --for production
```

**WARNING**

> authorization-related annotations in your CDS models. Ensure to rerun `cds compile --to xsuaa`, as documented in the *Authorization* guide whenever there are changes to these annotations.

▶ *For trial and extension landscapes, OAuth configuration is required*

↳ *Learn more about SAP Authorization and Trust Management/XSUAA.*

## 3. Using MTA-Based Deployment

We'll be using the Cloud MTA Build Tool to execute the deployment. The modules and services are configured in an `mta.yaml` deployment descriptor file, which we generate with:

```sh
cds add mta
```

↳ *Learn more about MTA-based deployment.*

## 4. Using App Router as Gateway

The *App Router* acts as a single point-of-entry gateway to route requests to. In particular, it ensures user login and authentication in combination with XSUAA.

Two deployment options are available:

- **Managed App Router**: for scenarios where *SAP Fiori Launchpad* (FLP) is the entry point to access your applications, the Managed App Router provided by SAP Fiori Launchpad is available. See the end-to-end tutorial for the necessary configuration in `mta.yaml` and on each *SAP Fiori application*.
- **Custom App Router**: for scenarios without SAP Fiori Launchpad, the App Router needs to be deployed along with your application. Use the following command to enhance the application configuration:

  ```sh
  cds add approuter --for production
  ```

↳ *Learn more about the SAP BTP Application Router.*

To enable multitenancy for production, run the following command:

```sh
cds add multitenancy --for production
```

> If necessary, modifies deployment descriptors such as *mta.yaml* for Cloud Foundry.

↳ *Learn more about MTX services.*

▶ *SaaS Service Dependencies*

> **You're set!**
>
> The previous steps are required *only once* in a project's lifetime. With that done, we can repeatedly deploy the application.

## 6. Freeze Dependencies

Deployed applications should freeze all their dependencies, including transient ones. Create a *package-lock.json* file for that:

```sh
npm update --package-lock-only
```

If you use multitenancy, also freeze dependencies for the MTX sidecar:

```sh
npm update --package-lock-only --prefix mtx/sidecar
```

↳ *Learn more about dependency management for Node.js*

> **Regularly update your `package-lock.json` to consume latest versions and bug fixes**
>
> Do so by running this command again, for example each time you deploy a new version of your application.

## Build Deployables with *cds build*

Run *cds build* to generate additional deployment artifacts and prepare everything for production in a local *./gen* folder as a staging area. While *cds build* is included in the next step *mbt build*, you can also run it selectively as a test, and to inspect what is generated:

```sh
cds build --production
```

↳ *Learn more about running and customizing cds build.*

## Assemble with *mbt build*

**Prepare monorepo setups**

The CAP samples repository on GitHub has a more advanced (monorepo) structure, so tell the *mbt* tool to find the *package-lock.json* on top-level:

```sh
ln -sf ../package-lock.json
```

Now, we use the *mbt* build tool to assemble everything into a single *mta.tar* archive:

```sh
mbt build -t gen --mtar mta.tar
```

↳ *Got errors? See the troubleshooting guide.*

↳ *Learn how to reduce the MTA archive size during development.*

# Deploy to Cloud

Finally, we can deploy the generated archive to Cloud Foundry:

↳ *You need to be logged in to Cloud Foundry.*

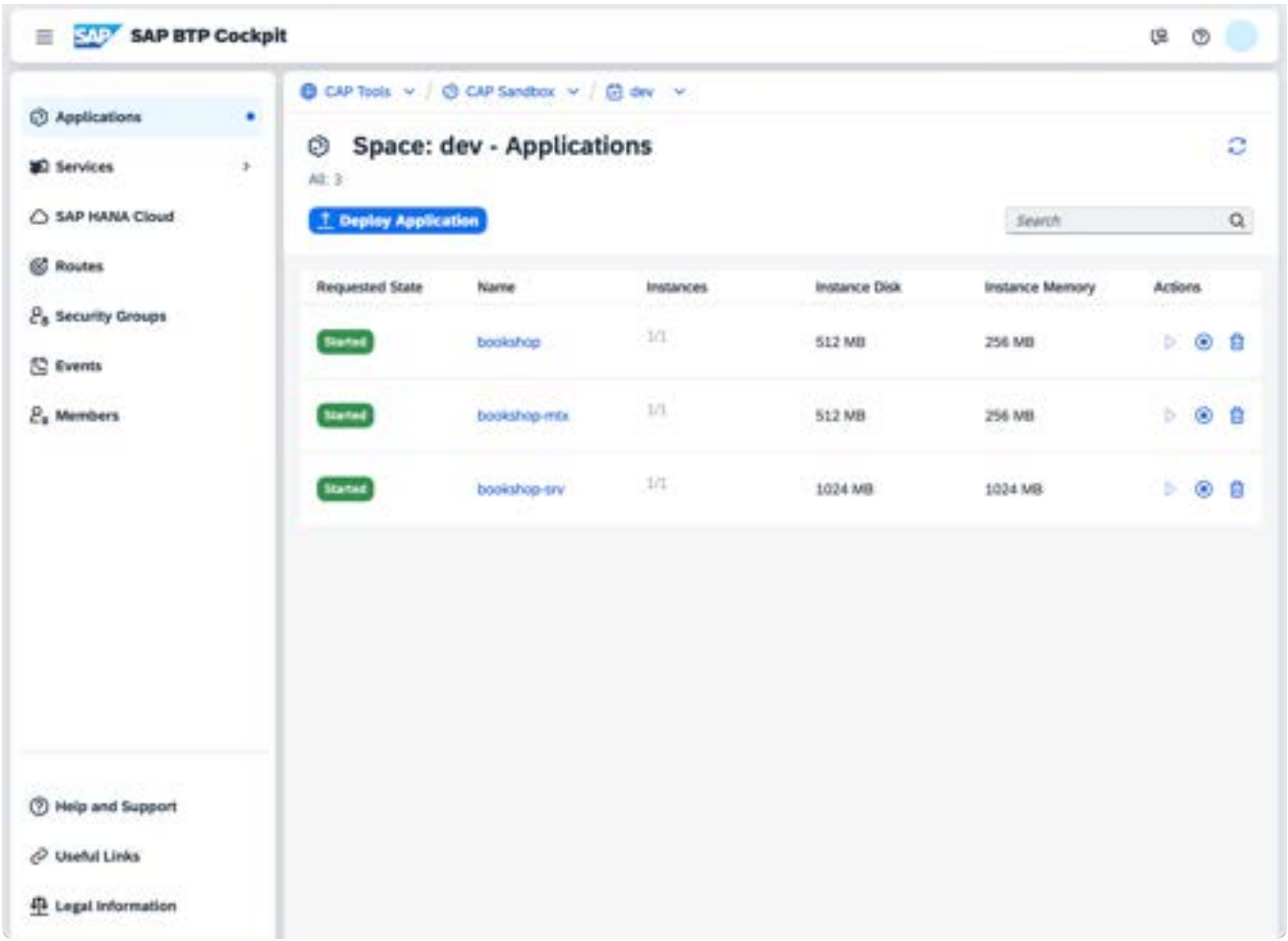
This process can take some minutes and finally creates a log output like this:

log
```
[…]
Application "bookshop" started and available at
"[org]-[space]-bookshop.landscape-domain.com"
[…]
```

Copy and open this URL in your web browser. It's the URL of your App Router application.

## Inspect Deployed Apps in BTP Cockpit

Visit the "Applications" section in your SAP BTP cockpit to see the deployed apps:




**Assign the *admin* role**

collection and assign the role and your user to get access.

↳ *Got errors? See the troubleshooting guide.*

# Appendices

## Deploy using `cf push`

As an alternative to MTA-based deployment, you can choose Cloud Foundry-native deployment using `cf push`, or `cf create-service-push` respectively.

### Prerequisites

Install the *Create-Service-Push* plugin:

```sh
cf install-plugin Create-Service-Push
```

This plugin acts the same way as `cf push`, but extends it such that services are *created* first. With the plain `cf push` command, this is not possible.

### Add a `manifest.yml`

```sh
cds add cf-manifest
```

This creates two files, a *manifest.yml* and *services-manifest.yml* in the project root folder.

server holding the service implementations, and the other one is a 'DB deployer' application, whose sole purpose is to start the SAP HANA deployment.

- *services-manifest.yml* defines which Cloud Foundry services shall be created. The services are derived from the service bindings in *package.json* using the `cds.requires` configuration.

> **Version-control manifest files**
>
> Unlike the files in the *gen* folders, these manifest files are genuine sources and should be added to the version control system. This way, you can adjust them to your needs as you evolve your application.

## Build the Project

This prepares everything for deployment, and -- by default -- writes the build output, that is the deployment artifacts, to folder *./gen* in your project root.

```sh
cds build --production
```

↳ *Learn how `cds build` can be configured.*

The `--production` parameter ensures that the cloud deployment-related artifacts are created by `cds build` . See section SAP HANA database deployment for more details.

## Push the Application

This command creates service instances, pushes the applications and binds the services to the application with a single call:

```sh
cf create-service-push
```

During deployment, the plugin reads the *services-manifest.yml* file and creates the services listed there. It then reads *manifest.yml*, pushes the applications defined there, and binds these applications to service instances created before. If the service instances already exist, only the `cf push` operation will be executed.

You can also apply some shortcuts:

deploy a single application.

- Use `cf create-service-push --no-push` to only create or update service-related data without pushing the applications.

In the deployment log, find the application URL in the `routes` line at the end:

```log
name:              bookshop-srv
requested state:   started
routes:            bookshop-srv.cfapps.sap.hana.ondemand.com
```

Open this URL in the browser and try out the provided links, for example, `…/browse/Books`. Application data is fetched from SAP HANA.

> **Ensure successful SAP HANA deployment**
>
> Check the deployment logs of the database deployer application using
>
> ```sh
> cf logs <app-name>-db-deployer --recent
> ```
>
> to ensure that SAP HANA deployment was successful. The application itself is by default in state `started` after HDI deployment has finished, even if the HDI deployer returned an error. To save resources, you can explicitly stop the deployer application afterwards.

> **No Fiori preview in the cloud**
>
> The SAP Fiori Preview, that you are used to see from local development, is only available for the development profile and not available in the cloud. For productive applications, you should add a proper SAP Fiori application.

> **WARNING**
>
> Multitenant applications are not supported yet as multitenancy-related settings are not added to the generated descriptors. The data has to be entered manually.

↳ *Got errors? See the troubleshooting guide.*

Deployment

Deploy to Kyma/K8s

# Serving Fiori UIs

CAP provides out-of-the-box support for SAP Fiori elements front ends.

This guide explains how to add one or more SAP Fiori elements apps to a CAP project, how to add SAP Fiori elements annotations to respective service definitions, and more. In the following sections, when mentioning Fiori, we always mean SAP Fiori elements.

↳ *Learn more about developing SAP Fiori elements and OData V4 (since 1.84.)*

Table of Contents


- SAP Fiori Preview
- Adding Fiori Apps
  - Using SAP Fiori Tools
  - From cap/samples
  - From Incidents Sample
- Fiori Annotations
  - What Are SAP Fiori Annotations?
  - Where to Put Them?
  - Maintaining Annotations
  - Code Completion
  - Diagnostics
  - Navigation to Referenced Annotations
  - Documentation (Quick Info)
  - Prefer @title and @description
  - Prefer @readonly, @mandatory, ...
- Draft Support
  - Enabling Draft with @odata.draft.enabled

    - Enabling Draft for Localized Data
    - Validating Drafts
    - Query Drafts Programmatically
- Use Roles to Toggle Visibility of UI elements
- Value Helps
    - Convenience Option @cds.odata.valuelist
    - Pre-Defined Types in @sap/cds/common
    - Usages of @sap/cds/common
    - Resulting Annotations in EDMX
- Actions
- Cache Control

## SAP Fiori Preview

For entities exposed via OData V4 there is a *Fiori preview* link on the index page. It dynamically serves an SAP Fiori Elements list page that allows you to quickly see the effect of annotation changes without having to create a UI application first.

▶ *Be aware that this is **not meant for production**.*

## Adding Fiori Apps

As showcased in cap/samples , SAP Fiori apps should be added as sub folders to the *app/* of a CAP project. Each sub folder constitutes an individual SAP Fiori application, with local annotations, *manifest.json*, etc. So, a typical folder layout would look like this:

| Folder/Sub Folder | Description |
| --- | --- |
| *app/* | All SAP Fiori apps should go in here |

| | |
|---|---|
| *browse/* | SAP Fiori app for end users |
| *orders/* | SAP Fiori app for order management |
| *admin/* | SAP Fiori app for admins |
| *index.html* | For sandbox tests |
| *srv/* | All services |
| *db/* | Domain models, and db stuff |

> **TIP**
>
> Links to Fiori applications created in the *app/* folder are automatically added to the index page of your CAP application for local development.

## Using SAP Fiori Tools

The SAP Fiori tools provide advanced support for adding SAP Fiori apps to existing CAP projects as well as a wealth of productivity tools, for example for adding SAP Fiori annotations, or graphical modeling and editing. They can be used locally in Visual Studio Code (VS Code) or in SAP Business Application Studio .

↳ *Learn more about **how to install SAP Fiori tools.***

## From cap/samples

For example, you can copy the SAP Fiori apps from cap/samples as a template and modify the content as appropriate.

## From Incidents Sample

This is a sample to create an incident management app with SAP Fiori elements for OData V4.

The main content to add is service definitions annotated with information about how to render respective data.

## What Are SAP Fiori Annotations?

SAP Fiori elements apps are generic front ends, which construct and render the pages and controls based on annotated metadata documents. The annotations provide semantic annotations used to render such content, for example:

```cds
annotate CatalogService.Books with @(
  UI: {
    SelectionFields: [ ID, price, currency_code ],
    LineItem: [
      {Value: title},
      {Value: author, Label:'{i18n>Author}'},
      {Value: genre.name},
      {Value: price},
      {Value: currency.symbol, Label:' '},
    ]
  }
);
```

↳ *Find this source and many more in **cap/samples**.*
↳ *Learn more about **OData Annotations in CDS**.*

## Where to Put Them?

While CDS in principle allows you to add such annotations everywhere in your models, we recommend putting them in separate *.cds* files placed in your *./app/** folders, for example, as follows.

```sh
./app  #> all your Fiori annotations should go here, for example:
   ./admin
      fiori-service.cds #> annotating ../srv/admin-service.cds
   ./browse
      fiori-service.cds #> annotating ../srv/cat-service.cds
```

```
./srv  #> all service definitions should stay clean in here:
  admin-service.cds
  cat-service.cds
...
```

↳ *See this also in **cap/samples/fiori**.*

**Reasoning:** This recommendation essentially follows the best practices and guiding principles of Conceptual Modeling and Separation of Concerns.

## Maintaining Annotations

Maintaining OData annotations in *.cds* files is accelerated by the SAP Fiori tools - CDS OData Language Server @sap/ux-cds-odata-language-server-extension in the SAP CDS language support plugin . It helps you add and edit OData annotations in CDS syntax with:

- Code completion
- Validation against the OData vocabularies and project metadata
- Navigation to the referenced annotations
- Quick view of vocabulary information
- Internationalization support

These assisting features are provided for OData annotations in CDS syntax and can't be used yet for the core data services common annotations.

The @sap/ux-cds-odata-language-server-extension module doesn't require any manual installation. The latest version is fetched by default from npmjs.com as indicated in the user preference setting **CDS > Contributions: Registry**.

↳ *Learn more about the **CDS extension for VS Code**.*

## Code Completion

The CDS OData Language Server provides a list of context-sensitive suggestions based on the service metadata and OData vocabularies. You can use it to choose OData annotation terms, their properties, and values from the list of suggestions in annotate directives applied to service entities and entity elements. See annotate directives for more details.

### Using Code Completion

list of suggested values is displayed.

> Note: You can filter the list of suggested values by typing more characters.

Navigate to the desired value using the up or down arrows or your mouse. Accept the highlighted value by pressing `Enter` or by clicking the mouse. Use code completion to add and change individual values (word-based completion) and to add small code blocks containing annotation structures along with mandatory properties (micro-snippets). In an active code snippet, you can use the `⇥` (tab) key to quickly move to the next tab stop.

**Example: Annotating Service Entities**

(cursor position indicated by `|`)

1. Place cursor in the *`annotate`* directive for a service entity, for example *`annotate Foo.Bar with ;`* and trigger code completion.
2. Type *`u`* to filter the suggestions and choose *`{} UI`*. Micro-snippet *`@UI : {|}`* is inserted: *`annotate Foo.Bar with @UI : {|};`*
3. Use code completion again to add an annotation term from the UI vocabulary, in this example *`SelectionFields`*. The micro snippet for this annotation is added and the cursor is placed directly after the term name letting you define a qualifier: *`annotate Foo.Bar with @UI : {SelectionFields | : []};`*
4. Press the `⇥` (tab) key to move the cursor to the next tab stop and use code completion again to add values. Because the *`UI.SelectionFields`* annotation is a collection of entity elements (entity properties), all elements of the annotated entity are suggested.

   > **TIP**
   >
   > To choose an element of an associated entity, first select the corresponding association from the list and type *. (period)*. Elements of associated entity are suggested.
   >
   > Note: You can add multiple values separated by comma.

```cds
annotate Foo.Bar with @UI : { SelectionFields : [
    description, assignedIndividual.lastName|
  ],
};
```

5. Add a new line after `,` (comma) and use code completion again to add another annotation from the UI vocabulary, such as *`LineItem`*. Line item is a collection of

completion list.

```cds
annotate Foo.Bar with @UI : {
   SelectionFields : [
     description, assignedIndividual.lastName
   ],
   LineItem : [{
     $Type:'UI.DataField',
     Value : |,
  },
};
```

> Note: For each record type, two kinds of micro-snippets are provided: one containing only mandatory properties and one containing all properties defined for this record (full record). Usually you need just a subset of properties. So, you either select a full record and then remove the properties you don't need, or add the record containing only required properties and then add the remaining properties.

6. Use code completion to add values for the annotation properties.

```cds
annotate Foo.Bar with @UI : {
  SelectionFields : [
    description, assignedIndividual.lastName
  ],
  LineItem : [
    {
        $Type:'UI.DataField',
        Value : description,
    },
    {
        $Type:'UI.DataFieldForAnnotation',
        Target :  'assignedIndividual/@Communication.Contact',
    },|
  ]
};
```

> Note: To add values pointing to annotations defined in another CDS source, you must reference this source with the *using* directive. See The *using* Directive for more details.

**Example: Annotating Entity Elements**

(cursor position indicated by | )

`{|};` , add a new line and trigger code completion. You get the list of entity elements. Choose the one that you want to annotate.

```cds
annotate Foo.Bar with {
    code|
};
```

2. Press the ␣ key, use code completion again, and choose `{} UI` . The `@UI : {|}` micro-snippet is inserted:

```cds
annotate Foo.Bar with {
    code @UI : { | }
};
```

3. Trigger completion again and choose an annotation term from the UI vocabulary, in this example: **Hidden**.

```cds
annotate Foo.Bar with {
    code @UI : {Hidden : |}
};
```

4. Press the ⇥ (tab) key to move the cursor to the next tab stop and use code completion again to add the value. Because the `UI.Hidden` annotation is of Boolean type, the values true and false are suggested:

```cds
annotate Foo.Bar with {
    code @UI : {Hidden : false }
};
```

## Diagnostics

The CDS OData Language Server validates OData annotations in *.cds* files against the service metadata and OData vocabularies. It also checks provided string content for language-dependent annotation values and warns you if the format doesn't match the internationalization (i18n) key reference. It shows you that this string is hard coded and won't change based on the language setting in your application. See Internationalization support for more details.

to the relevant files.

You can view the diagnostic messages by hovering over the highlighted part in the annotation file or by opening the problems panel. Click on the message in the problems panel to navigate to the related place in the annotation file.

> Note: If an annotation value points to the annotation defined in another CDS source, you must reference this source with a *`using`* directive to avoid warnings. See The *`using`* Directive for more details.

## Navigation to Referenced Annotations

CDS OData Language Server enables quick navigation to the definition of referenced annotations. For example, if your annotation file contains a *`DataFieldForAnnotation`* record referencing an *`Identification`* annotation defined in the service file, you can view which file it's defined in and what fields or labels this annotation contains. You can even update the *`Identification`* annotation or add comments.

You can navigate to the referenced annotation using the Peek Definition and Go To Definition features.

> Note: If the referenced annotation is defined in another CDS source, you must reference this source with the *`using`* directive to enable the navigation. See The *`using`* Directive for more details.

### Peek Definition

Peek Definition lets you preview and update the referenced annotation without switching away from the code that you're writing. It's triggered when your cursor is inside the referenced annotation value.

- Using a keyboard: choose `⌥` + `F12` (macOS) or `Alt` + `F12` (other platforms)
- Using a mouse: right-click and select **Peek Definition** If an annotation is defined in multiple sources, all these sources are listed. You can select which one you want to view or update. Annotation layering isn't considered.

### Go to Definition

Go To Definition lets you navigate to the source of the referenced annotation and opens the source file scrolled to the respective place in a new tab. It's triggered when your cursor is inside the referenced annotation value.

string value, and trigger Go to Definition:

- Using a keyboard: choose `F12` in VS Code, or `Ctrl` + `F12` in SAP Business Application Studio
- Using a mouse: right-click and select **Go To Definition**
- Using a keyboard and mouse: `⌘` + mouse click (macOS) or `Ctrl` + mouse click (other platforms)

If an annotation is defined in multiple sources, a Peek definition listing these sources will be shown instead. Annotation layering isn't considered.

## Documentation (Quick Info)

The annotation language server provides quick information for annotation terms, record types, and properties used in the annotation file, or provided as suggestions in code completion lists. This information is retrieved from the respective OData vocabularies and can provide answers to the following questions:

- What is the type and purpose of the annotation term/record type/property?
- What targets can the annotation term apply to?
- Is the annotation term/record type/property experimental? Is it deprecated?
- Is this annotation property mandatory or optional?

> Note: The exact content depends on the availability in OData vocabularies.

To view the quick info for an annotation term, record type, or property used in the annotation file, hover your mouse over it. The accompanying documentation is displayed in a hover window, if provided in the respective OData vocabularies.

To view the quick info for each suggestion in the code completion list, either pressing `⌘` + `␣` (macOS) or `Ctrl` + `␣` (other platforms), or click the *info* icon. The accompanying documentation for the suggestion expands to the side. The expanded documentation stays open and updates as you navigate the list. You can close this by pressing `⌘` + `␣` / `Ctrl` + `␣` again or by clicking on the close icon.

### Internationalization Support

When you open an annotation file, all language-dependent string values are checked against the *i18n.properties* file. Each value that doesn't represent a valid reference to the existing text key in the *i18n.properties* file, is indicated with a warning. A Quick Fix action is

reference to that entry.

## Prefer *@title* and *@description*

Influenced by the JSON Schema , CDS supports the common annotations *@title* and *@description* , which are mapped to corresponding OData annotations as follows:

| CDS | JSON Schema | OData |
|---|---|---|
| *@title* | *title* | *@Common.Label* |
| *@description* | *description* | *@Core.Description* |

We recommend preferring these annotations over the OData ones in protocol-agnostic data models and service models, for example:

cds
```
annotate my.Books with { //...
   title @title: 'Book Title';
   author @title: 'Author ID';
}
```

## Prefer *@readonly* , *@mandatory* , ...

CDS supports *@readonly* as a common annotation, which translates to respective OData annotations from the *@Capabilities* vocabulary. We recommend using the former for reasons of conciseness and comprehensibility as shown in this example:

cds
```
@readonly entity Foo {   // entity-level
  @readonly foo : String // element-level
}
```

is equivalent to:

cds
```
entity Foo @(Capabilities:{
  // entity-level
  InsertRestrictions.Insertable: false,
  UpdateRestrictions.Updatable: false,
  DeleteRestrictions.Deletable: false
```

```
  // element-level
  @Core.Computed foo : String
}
```

Similar recommendations apply to `@mandatory` and others → see Common Annotations.

---

## Draft Support

SAP Fiori supports edit sessions with draft states stored on the server, so users can interrupt and continue later on, possibly from different places and devices. CAP, as well as SAP Fiori elements, provide out-of-the-box support for drafts as outlined in the following sections. **We recommend to always use draft** when your application needs data input by end users.

↳ *For details and guidelines, see **SAP Fiori Design Guidelines for Draft**.*

↳ *Find a working end-to-end version in **cap/samples/fiori**.*

↳ *For details about the draft flow in SAP Fiori elements, see **SAP Fiori elements > Draft Handling***

### Enabling Draft with `@odata.draft.enabled`

To enable draft for an entity exposed by a service, simply annotate it with `@odata.draft.enabled` as in this example:

```cds
annotate AdminService.Books with @odata.draft.enabled;
```

↳ *See it live in **cap/samples**.*

> **WARNING**
>
> You can't project from draft-enabled entities, as annotations are propagated. Either *enable* the draft for the projection and not the original entity or *disable* the draft on the projection using `@odata.draft.enabled: null` .

Be aware that you must not modify associated entities through drafts. Only compositions will get a "Create" button in SAP Fiori elements UIs because they are stored as part of the same draft entity.

## Enabling Draft for Localized Data

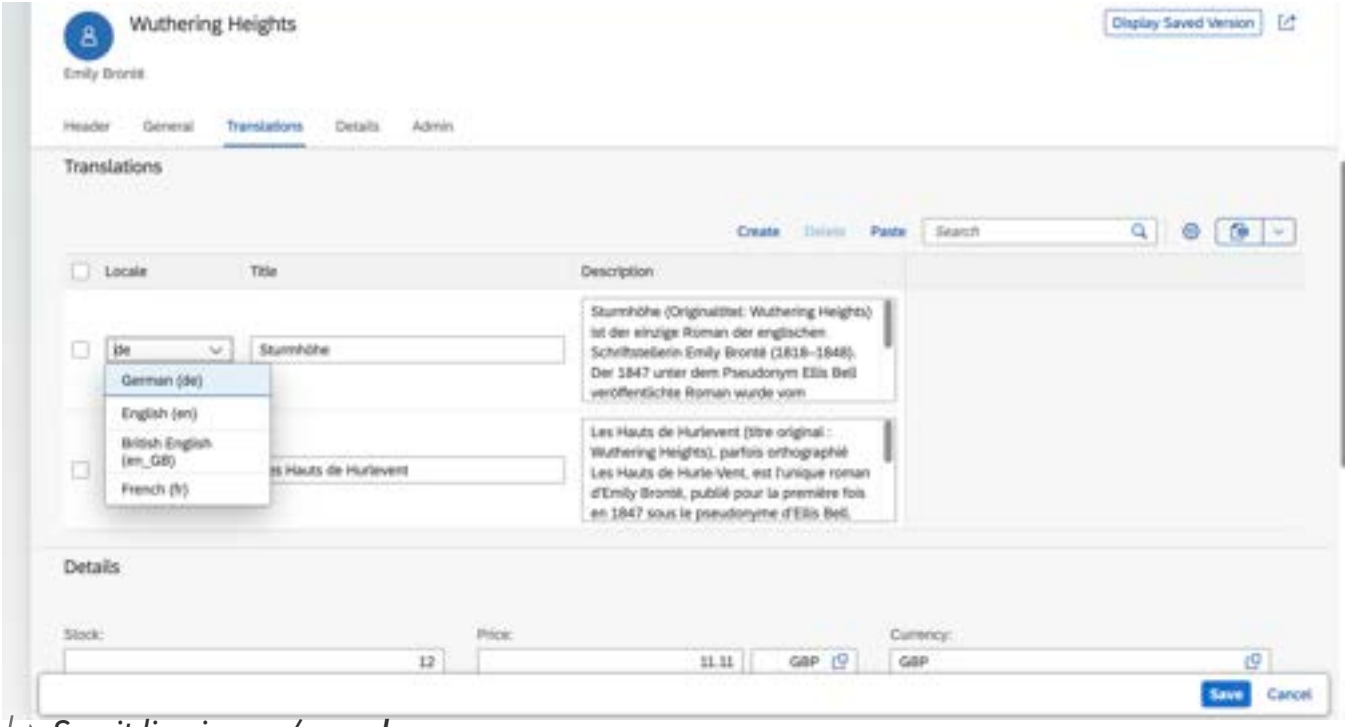
Annotate the underlying base entity in the base model with `@fiori.draft.enabled` to also support drafts for localized data:

```cds
annotate sap.capire.bookshop.Books with @fiori.draft.enabled;
```

#### Background

SAP Fiori drafts required single keys of type `UUID`, which isn't the case by default for the automatically generated `_texts` entities (→ see the *Localized Data* guide for details). The `@fiori.draft.enabled` annotation tells the compiler to add such a technical primary key element named `ID_texts`.

> **WARNING**
>
> Adding the annotation `@fiori.draft.enabled` won't work if the corresponding `_texts` entities contain any entries, because existing entries don't have a value for the new key field `ID_texts`.




↳ *See it live in cap/samples.*

If you're editing data in multiple languages, the *General* tab in the example above is reserved for the default language (often "en"). Any change to other languages has to be done in the *Translations* tab, where a corresponding language can be chosen from a drop-down menu as illustrated above. This also applies if you use the URL parameter `sap-language` on the draft page.

## Validating Drafts

You can add custom handlers to add specific validations, as usual. In addition, for a draft, you can register handlers to the `PATCH` events to validate input per field, during the edit session, as follows.

**... in Java**

You can add your validation logic before operation event handlers. Specific events for draft operations exist. See Java > Fiori Drafts > Editing Drafts for more details.

**... in Node.js**

You can add your validation logic before the operation handler for either CRUD or draft-specific events. See Node.js > Fiori Support > Handlers Registration for more details about handler registration.

## Query Drafts Programmatically

```js
SELECT.from(Books.drafts) //returns all drafts of the Books entity
```

↳ *Learn how to query drafts in Java.*

## Use Roles to Toggle Visibility of UI elements

In addition to adding restrictions on services, entities, and actions/functions, there are use cases where you only want to hide certain parts of the UI for specific users. This is possible by using the respective UI annotations like `@UI.Hidden` or `@UI.CreateHidden` in conjunction with `$edmJson` pointing to a singleton.

First, you define the singleton in your service and annotate it with `@cds.persistency.skip` so that no database artefact is created:

```cds
@odata.singleton @cds.persistency.skip
entity Configuration {
    key ID: String;
    isAdmin : Boolean;
}
```

> A key is technically not required, but without it some consumers might run into problems.

Then define an `on` handler for serving the request:

```js
srv.on('READ', 'Configuration', async req => {
    req.reply({
        isAdmin: req.user.is('admin') //admin is the role, which for example
    });
});
```

Finally, refer to the singleton in the annotation by using a dynamic expression:

```cds
annotate service.Books with @(
    UI.CreateHidden : { $edmJson: {$Not: { $Path: '/CatalogService.EntityCont
```

```
);
```

The Entity Container is OData specific and refers to the `$metadata` of the OData service in which all accessible entities are located within the Entity Container.

▶ *SAP Fiori elements also allows to not include it in the path*

---

## Value Helps

In addition to supporting the standard `@Common.ValueList` annotations as defined in the OData Vocabularies, CAP provides advanced, convenient support for Value Help as understood and supported by SAP Fiori.

### Convenience Option `@cds.odata.valuelist`

Simply add the `@cds.odata.valuelist` annotation to an entity, and all managed associations targeting this entity will automatically receive Value Lists in SAP Fiori clients. For example:

cds

```
@cds.odata.valuelist
entity Currencies {}
```

cds

```
service BookshopService {
   entity Books { //...
      currency : Association to Currencies;
   }
}
```

### Pre-Defined Types in `@sap/cds/common`

all uses automatically benefit from this. This is an effective excerpt of respective definitions in *@sap/cds/common* :

```cds
type Currencies : Association to sap.common.Currencies;
```

```cds
context sap.common {
  entity Currencies : CodeList {...};
  entity CodeList { name : localized String; ... }
}
```

```cds
annotate sap.common.CodeList with @(
   UI.Identification: [name],
   cds.odata.valuelist,
);
```

## Usages of *@sap/cds/common*

In effect, usages of @sap/cds/common stay clean of any pollution, for example:

```cds
using { Currency } from '@sap/cds/common';
entity Books { //...
  currency : Currency;
}
```

↳ *Find this also in our **cap/samples**.*

Still, all SAP Fiori UIs, on all services exposing *Books* , will automatically receive Value Help for currencies. You can also benefit from that when deriving your project-specific code list entities from **sap.common.CodeList**.

## Resulting Annotations in EDMX

Here is an example showing how this ends up as OData *Common.ValueList* annotations:

```xml
<Annotations Target="AdminService.Books/currency_code">
  <Annotation Term="Common.ValueList">
    <Record Type="Common.ValueListType">
      <PropertyValue Property="CollectionPath" String="Currencies"/>
```

```
      <PropertyValue Property="Parameters">
        <Collection>
          <Record Type="Common.ValueListParameterInOut">
            <PropertyValue Property="ValueListProperty" String="code"/>
            <PropertyValue Property="LocalDataProperty" PropertyPath="currenc
          </Record>
          <Record Type="Common.ValueListParameterDisplayOnly">
            <PropertyValue Property="ValueListProperty" String="name"/>
          </Record>
        </Collection>
      </PropertyValue>
    </Record>
  </Annotation>
</Annotation>
```

## Actions

In our SFLIGHT sample application, we showcase how to use actions covering the definition in your CDS model, the needed custom code and the UI implementation.

↳ *Learn more about Custom Actions & Functions.*

We're going to look at three things.

1. Define the action in CDS and custom code.
2. Create buttons to bring the action to the UI
3. Dynamically define the buttons status on the UI

First you need to define an action, like in the *travel-service.cds* file .

cds

```
entity Travel as projection on my.Travel actions {
    action createTravelByTemplate() returns Travel;
    action rejectTravel();
    action acceptTravel();
    action deductDiscount( percent: Percentage not null ) returns Travel;
};
```

*travel-service.js* file for example:

```js
this.on('acceptTravel', req => UPDATE(req._target).with({TravelStatus_code:'A
```

> Note: *req._target* is a workaround that has been introduced in SFlight. In the future, there might be an official API for it.

Create the buttons, to bring this action onto the UI and make it actionable for the user. There are two buttons: On the overview and in the detail screen. Both are defined in the *layouts.cds* file.

For the overview of all travels, use the *@UI.LineItem* annotation.

```cds
annotate TravelService.Travel with @UI : {
LineItem : [
    { $Type  : 'UI.DataFieldForAction',
      Action : 'TravelService.acceptTravel',
      Label  : '{i18n>AcceptTravel}'   }
  ]
};
```

For the detail screen of a travel, use the *@UI.Identification* annotation.

```cds
annotate TravelService.Travel with @UI : {
  Identification : [
    { $Type  : 'UI.DataFieldForAction',
      Action : 'TravelService.acceptTravel',
      Label  : '{i18n>AcceptTravel}'   }
  ]
};
```

Now, the buttons are there and connected to the action. The missing piece is to define the availability of the buttons dynamically. Annotate the *Travel* entity in the *TravelService* service accordingly in the *field-control.cds* file.

```cds
annotate TravelService.Travel with actions {
 acceptTravel @(
   Core.OperationAvailable : {
     $edmJson: { $Ne: [{ $Path: 'in/TravelStatus_code'}, 'A']}
```

```
Common.SideEffects.TargetProperties : ['in/TravelStatus_code'], } };
```

This annotation uses dynamic expressions to control the buttons for each action. And the status of a travel on the UI is updated, triggered by the `@Common.SideEffects.TargetProperties` annotation.

> **More complex calculation**
>
> If you have the need for a more complex calculation, then the interesting parts in SFLIGHT are virtual fields in *field-control.cds* (also lines 37-44) and custom code in *travel-service.js* .

## Cache Control

CAP provides the option to set a Cache-Control header with a max-age directive to indicate that a response remains fresh until *n* seconds after it was generated . In the CDS model, this can be done using the `@http.CacheControl: {maxAge: <seconds>}` annotation on stream properties. The header indicates that caches can store the response and reuse it for subsequent requests while it's fresh. The `max-age` (in seconds) specifies the maximum age of the content before it becomes stale.

> **Elapsed time since the response was generated**
>
> The `max-age` is the elapsed time since the response was generated on the origin server. It's not related to when the response was received.

> **Only Java**
>
> Cache Control feature is currently supported on the Java runtime only.

CAP Node.js 8 and CAP Java 3 available now

# Events and Messaging

CAP provides intrinsic support for emitting and receiving events. This is complemented by Messaging Services connecting to message brokers to exchange event messages across remote services.

Table of Contents


- Ubiquitous Events in CAP
  - Intrinsic Eventing in CAP Core
  - Typical Emitter and Receiver Roles
  - Ubiquitous Notion of Events
  - Asynchronous & Synchronous APIs
  - Why Using Messaging?
- Books Reviews Sample
  - Declaring Events in CDS
  - Emitting Events
  - Receiving Events
- In-Process Eventing
  - 1. Start a Single Server Process
  - 2. Add or Update Reviews
  - 3. Check Ratings in Bookshop App
- Using Message Channels
  - 1. Use file-based-messaging in Development
  - 2. Start the reviews Service and bookstore Separately
  - 3. Add or Update Reviews
  - 4. Shut Down and Restart Receiver → Resilience by Design

- Using Multiple Channels
  - Using Separate Channels
  - Using composite-messaging Implementation
  - Configuring Individual Channels and Routes
- Low-Level Messaging
- CloudEvents Standard
- Using SAP Event Mesh
- Events from SAP S/4HANA


---

# Ubiquitous Events in CAP

We're starting with an introduction to the core concepts in CAP. If you want to skip the introduction, you can fast-forward to the samples part starting at Books Reviews Sample.

## Intrinsic Eventing in CAP Core

As introduced in About CAP, everything happening at runtime is in response to events, and all service implementations take place in event handlers. All CAP services intrinsically support emitting and reacting to events, as shown in this simple code snippet (you can copy & run it in `cds repl`):

```js
let srv = new cds.Service
// Receiving Events
srv.on ('some event', msg => console.log('1st listener received:', msg))
srv.on ('some event', msg => console.log('2nd listener received:', msg))
// Emitting Events
await srv.emit ('some event', { foo:11, bar:'12' })
```

> **Intrinsic support for events**
>
> The core of CAP's processing model: all services are event emitters. Events can be sent to them, emitted by them, and event handlers register with them to react to such events.

In contrast to the previous code sample, emitters and receivers of events are decoupled, in different services and processes. And as all active things in CAP are services, so are usually emitters and receivers of events. Typical patterns look like that:

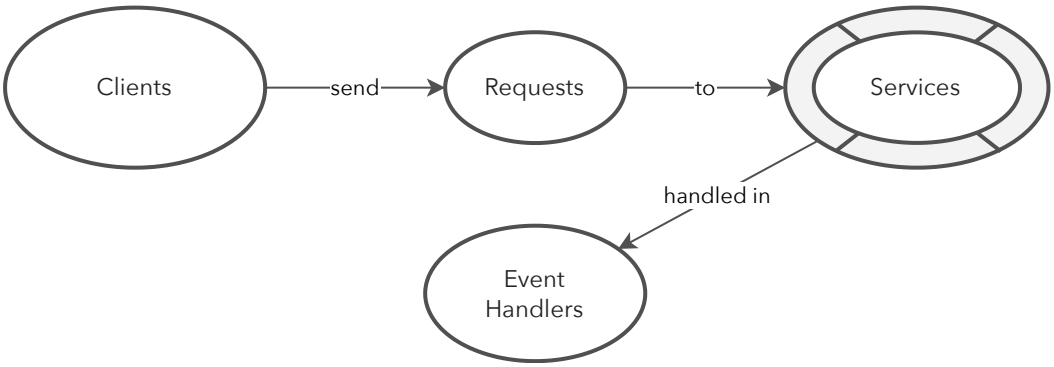
```js
class Emitter extends cds.Service { async someMethod() {
  // inform unknown receivers about something happened
  await this.emit ('some event', { some:'payload' })
}}
```

```js
class Receiver extends cds.Service { async init() {
  // connect to and register for events from Emitter
  const Emitter = await cds.connect.to('Emitter')
  Emitter.on ('some event', msg => {...})
}}
```

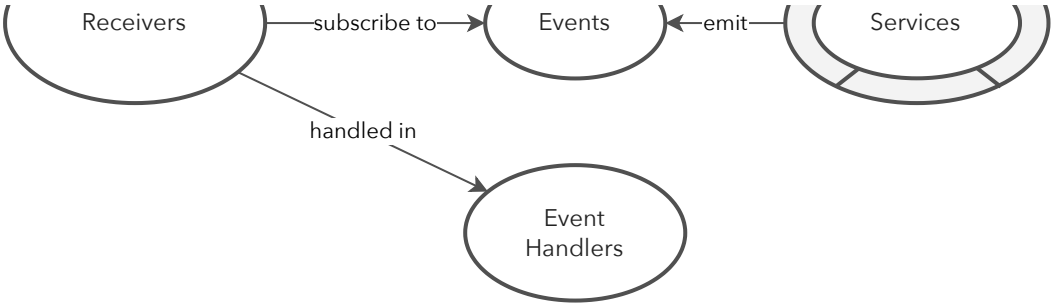
> **Emitters vs Receivers**
>
> **Emitters** usually emit messages to *themselves* to inform *potential* listeners about certain events. **Receivers** connect to *Emitters* to register handlers to such emitted events.

## Ubiquitous Notion of Events

A *Request* in CAP is actually a specialization of an *Event Message*. The same intrinsic mechanisms of sending and reacting to events are used for asynchronous communication in inverse order. A typical flow:




Asynchronous communication looks similar, just with reversed roles:

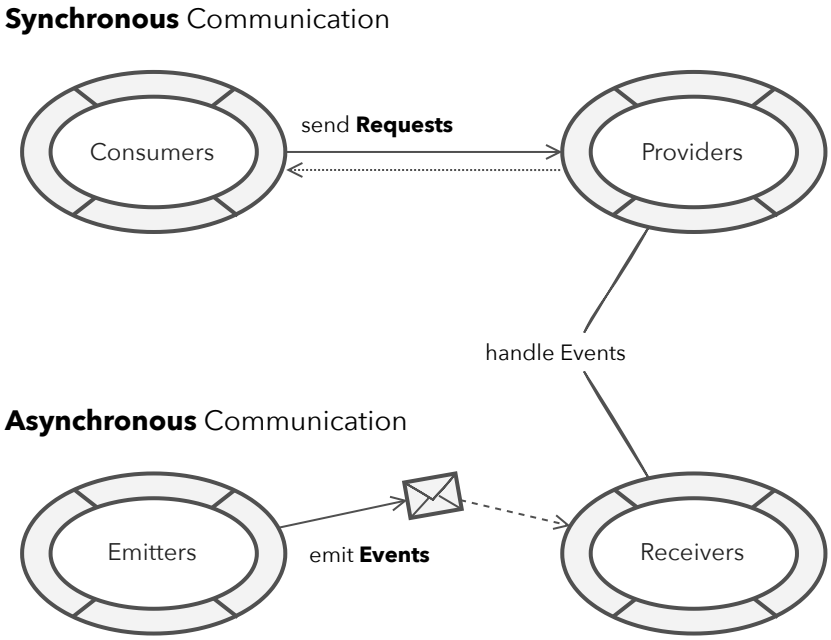



**Event Listeners vs Interceptors**

Requests are handled the same ways as events, with one major difference: While `on` handlers for events are *listeners* (all are called), handlers for synchronous requests are *interceptors* (only the topmost is called by the framework). An interceptor then decides whether to pass down control to `next` handlers or not.

## Asynchronous & Synchronous APIs

To sum up, handling events in CAP is done in the same way as you would handle requests in a service provider. Also, emitting event messages is similar to sending requests. The major difference is that the initiative is inverted: While *Consumers* connect to *Services* in synchronous communications, the *Receivers* connect to *Emitters* in asynchronous ones; *Emitters* in turn don't know *Receivers*.




**Blurring the line between synchronous and asynchronous API**

handlers for those events in order to implement the logic of how to react to these events.

### Why Using Messaging?

Using messaging has two major advantages:

**Resilience**

If a receiving service goes offline for a while, event messages are safely stored, and guaranteed to be delivered to the receiver as soon as it goes online again.

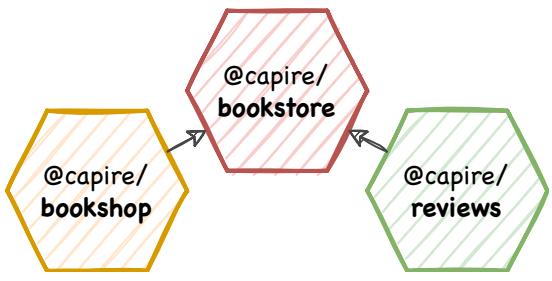
**Decoupling**

Emitters of event messages are decoupled from the receivers and don't need to know them at the time of sending. This way a service is able to emit events that other services can register on in the future, for example, to implement **extension** points.

## Books Reviews Sample

The following explanations walk us through a books review example from cap/samples:

- @capire/bookshop provides the well-known basic bookshop app.
- @capire/reviews provides an independent service to manage reviews.
- @capire/bookstore combines both into a composite application.




**TIP**

following steps.

## Declaring Events in CDS

Package *@capire/reviews* essentially provides a *ReviewsService*, declared like that:

cds

```
service ReviewsService {

  // Sync API
  entity Reviews as projection on my.Reviews excluding { likes }
  action like (review: Reviews:ID);
  action unlike (review: Reviews:ID);

  // Async API
  event reviewed : {
    subject : Reviews:subject;
    count   : Integer;
    rating  : Decimal; // new avg rating
  }

}
```

↳ *Learn more about declaring events in CDS.*

As you can read from the definitions, the service's synchronous API allows to create, read, update, and delete user *Reviews* for arbitrary review subjects. In addition, the service's asynchronous API declares the *reviewed* event that shall be emitted whenever a subject's average rating changes.

> **TIP**
>
> **Services in CAP** combine **synchronous** *and* **asynchronous** APIs. Events are declared on conceptual level focusing on domain, instead of low-level wire protocols.

## Emitting Events

Find the code to emit events in *@capire/reviews/srv/reviews-service.js*:

```
  // Emit a `reviewed` event whenever a subject's avg rating changes
  this.after (['CREATE','UPDATE','DELETE'], 'Reviews', (req) => {
    let { subject } = req.data, count, rating //= ...
    return this.emit ('reviewed', { subject, count, rating })
  })

}}
```

↳ *Learn more about `srv.emit()` in Node.js.*
↳ *Learn more about `srv.emit()` in Java.*

Method `srv.emit()` is used to emit event messages. As you can see, emitters usually emit messages to themselves, that is, `this`, to inform potential listeners about certain events. Emitters don't know the receivers of the events they emit. There might be none, there might be local ones in the same process, or remote ones in separate processes.

> **Messaging on Conceptual Level**
>
> Simply use `srv.emit()` to emit events, and let the CAP framework care for wire protocols like CloudEvents, transports via message brokers, multitenancy handling, and so forth.

## Receiving Events

Find the code to receive events in *@capire/bookstore/srv/mashup.js* (which is the basic bookshop app enhanced by reviews, hence integration with `ReviewsService`):

```
// Update Books' average ratings when reviews are updated
ReviewsService.on ('reviewed', (msg) => {
  const { subject, count, rating } = msg.data
  // ...
})
```

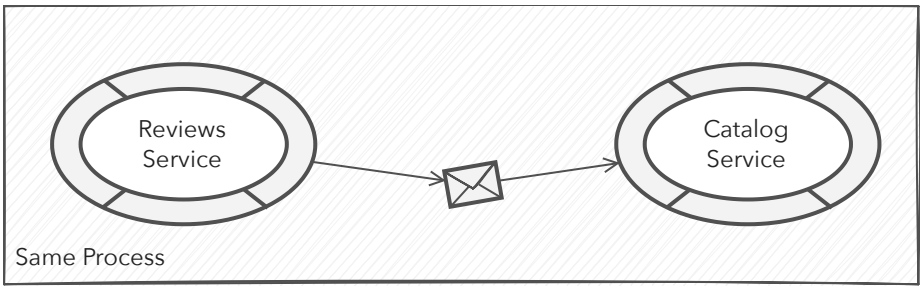
js

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

The message payload is in the `data` property of the inbound `msg` object.

## In-Process Eventing

As emitting and handling events is an intrinsic feature of the CAP core runtimes, there's nothing else required when emitters and receivers live in the same process.




Let's see that in action...

### 1. Start a Single Server Process

Run the following command to start a reviews-enhanced bookshop as an all-in-one server process:

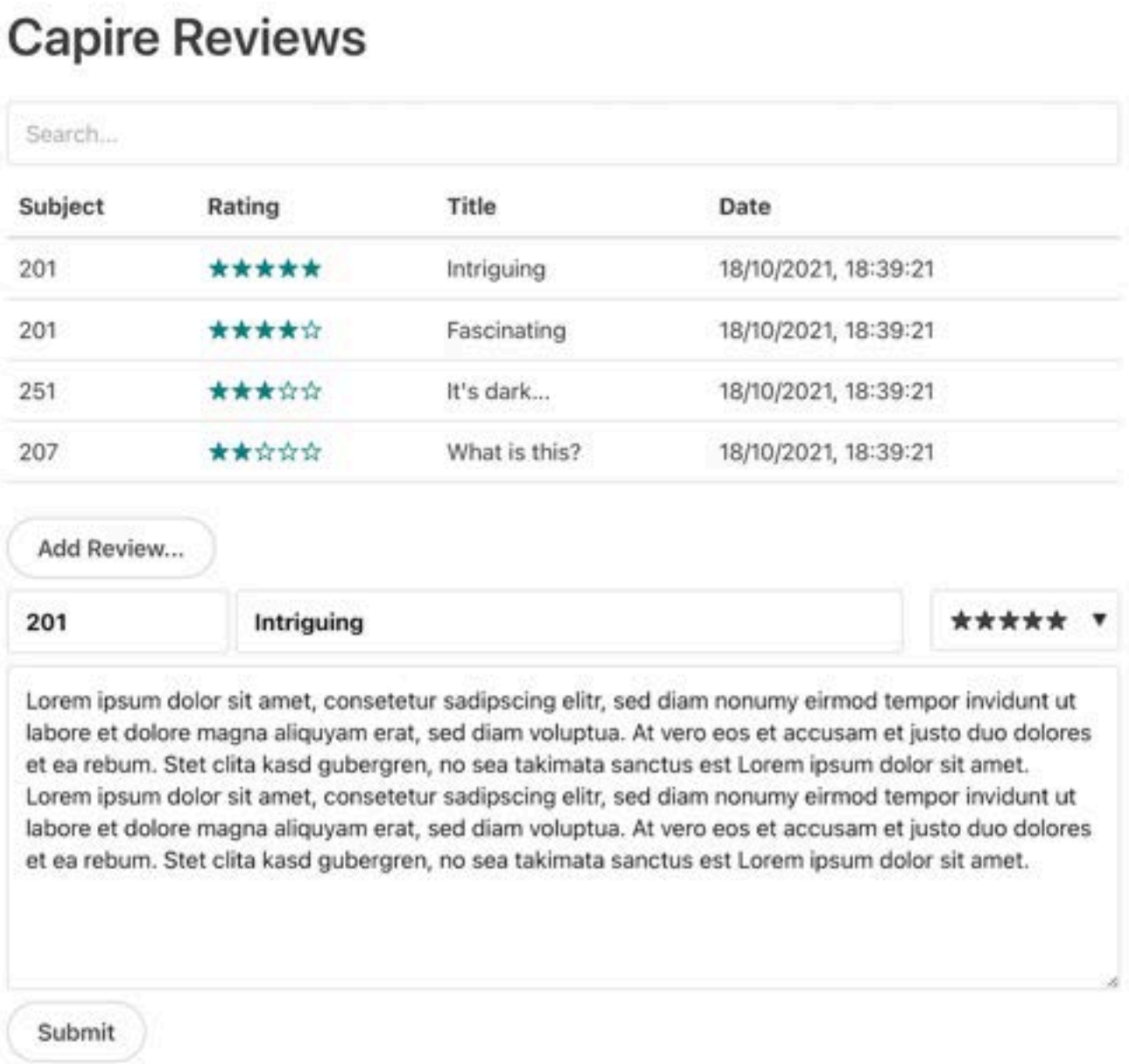
```sh
cds watch bookstore
```

It produces a trace output like that:

```log
[cds] - mocking ReviewsService { path: '/reviews', impl: '../reviews/srv/revi
[cds] - mocking OrdersService { path: '/orders', impl: '../orders/srv/orders-
[cds] - serving CatalogService { path: '/browse', impl: '../bookshop/srv/cat-
[cds] - serving AdminService { path: '/admin', impl: '../bookshop/srv/admin-s

[cds] - server listening on { url: 'http://localhost:4004' }
[cds] - launched at 5/25/2023, 4:53:46 PM, version: 7.0.0, in: 991.573ms
```

As apparent from the output, both, the two bookshop services `CatalogService` and `AdminService` as well as our new `ReviewsService`, are served in the same process (mocked, as the `ReviewsService` is configured as required service in *bookstore/package.json*).

Now, open http://localhost:4004/reviews to display the Vue.js UI that is provided with the reviews service sample:




- Choose one of the reviews.
- Change the 5-star rating with the dropdown.
- Choose *Submit*.
- Enter *bob* to authenticate.

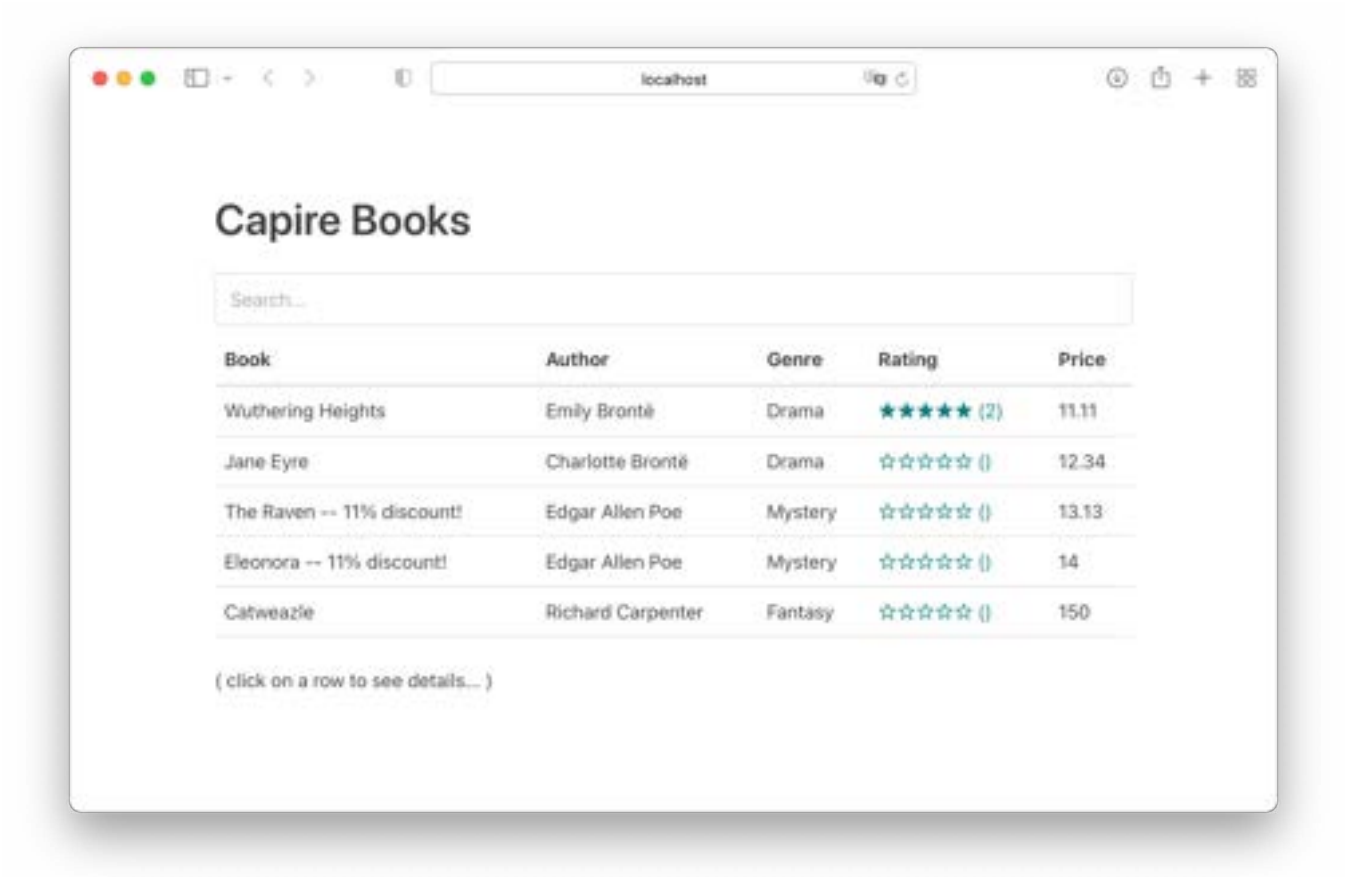
→ In the terminal window you should see a server reaction like this:

```log
[cds] - PATCH /reviews/Reviews/148ddf2b-c16a-4d52-b8aa-7d581460b431
< emitting: reviewed { subject: '201', count: 2, rating: 4.5 }
```

Which means the `ReviewsService` emitted a `reviewed` message that was received by the enhanced `CatalogService` .

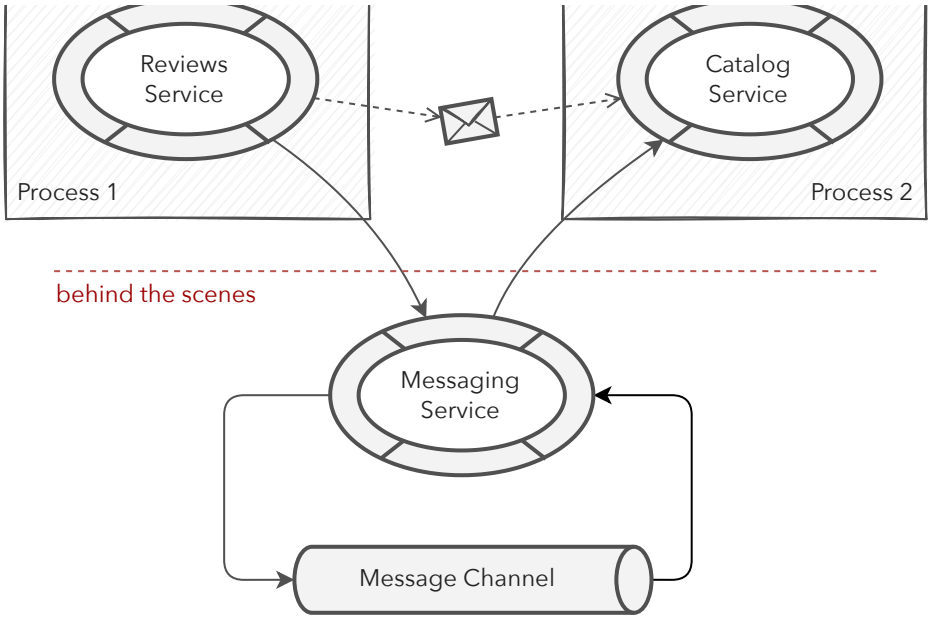
### 3. Check Ratings in Bookshop App

Open http://localhost:4004/bookshop to see the list of books served by `CatalogService` and refresh to see the updated average rating and reviews count:




## Using Message Channels

When emitters and receivers live in separate processes, you need to add a message channel to forward event messages. CAP provides messaging services, which take care for that message channel behind the scenes as illustrated in the following graphic:




**Uniform, Agnostic Messaging**

CAP provides messaging services, which transport messages behind the scenes using different messaging channels and brokers. All of this happens without the need to touch your code, which stays on conceptual level.

## 1. Use *`file-based-messaging`* in Development

For quick tests during development, CAP provides a simple file-based messaging service implementation. Configure that as follows for the *`[development]`* profile:

jsonc

```
"cds": {
  "requires": {
    "messaging": {
      "[development]": { "kind": "file-based-messaging" }
    },
  }
}
```

↳ *Learn more about `cds.env` profiles.*

In our samples, you find that in @capire/reviews/package.json as well as @capire/bookstore/package.json , which you'll run in the next step as separate processes.

First start the *reviews* service separately:

```sh
cds watch reviews
```

The trace output should contain these lines, confirming that you're using *file-based-messaging*, and that the *ReviewsService* is served by that process at port 4005:

```log
[cds] - connect to messaging > file-based-messaging { file: '~/.cds-msg-box'
[cds] - serving ReviewsService { path: '/reviews', impl: '../reviews/srv/revi

[cds] - server listening on { url: 'http://localhost:4005' }
[cds] - launched at 5/25/2023, 4:53:46 PM, version: 7.0.0, in: 593.274ms
```

Then, in a separate terminal start the *bookstore* server as before:

```sh
cds watch bookstore
```

This time the trace output is different to when you started all in a single server. The output confirms that you're using *file-based-messaging*, and that you now *connected* to the separately started *ReviewsService* at port 4005:

```log
[cds] - connect to messaging > file-based-messaging { file: '~/.cds-msg-box'
[cds] - mocking OrdersService { path: '/orders', impl: '../orders/srv/orders-
[cds] - serving CatalogService { path: '/browse', impl: '../reviews/srv/cat-s
[cds] - serving AdminService { path: '/admin', impl: '../reviews/srv/admin-se
[cds] - connect to ReviewsService > odata { url: 'http://localhost:4005/revie

[cds] - server listening on { url: 'http://localhost:4004' }
[cds] - launched at 5/25/2023, 4:55:46 PM, version: 7.0.0, in: 1.053s
```

## 3. Add or Update Reviews

Similar to before, open http://localhost:4005/vue/index.html to add or update reviews.

→ In the terminal window for the *reviews* server you should see this:

```
< emitting: reviewed { subject: '251', count: 1, rating: 3 }
```

→ In the terminal window for the *bookstore* server you should see this:

```
> received: reviewed { subject: '251', count: 1, rating: 3 }
```
log

> **Agnostic Messaging APIs**
>
> Without touching any code the event emitted from the *ReviewsService* got transported via *file-based-messaging* channel behind the scenes and was received in the *bookstore* as before, when you used in-process eventing → which was to be shown (*QED*).

## 4. Shut Down and Restart Receiver → Resilience by Design

You can simulate a server outage to demonstrate the value of messaging for resilience as follows:

1. Terminate the *bookstore* server with `Ctrl` + `C` in the respective terminal.
2. Add or update more reviews as described before.
3. Restart the receiver with *cds watch bookstore* .

→ You should see some trace output like that:

```
[cds] - server listening on { url: 'http://localhost:4004' }
[cds] - launched at 5/25/2023, 10:45:42 PM, version: 7.0.0, in: 1.023s
[cds] - [ terminate with ^C ]

> received: reviewed { subject: '207', count: 1, rating: 2 }
> received: reviewed { subject: '207', count: 1, rating: 2 }
> received: reviewed { subject: '207', count: 1, rating: 2 }
```
log

> **Resilience by Design**
>
> All messages emitted while the receiver was down stayed in the messaging queue and are delivered when the server is back.

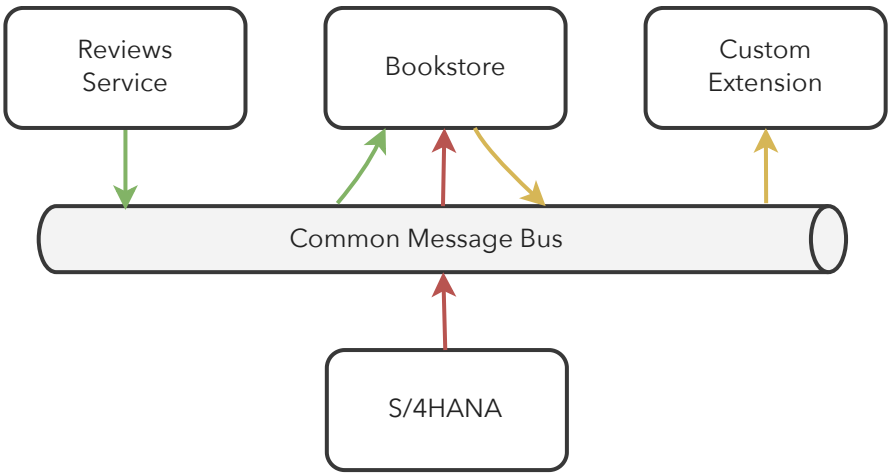
You can watch the messages flowing through the message queue by opening *~/.cds-msg-box* in a text editor. When the receiver is down and therefore the message not already consumed, you can see the event messages emitted by the `ReviewsService` in entries like that:

```json
ReviewsService.reviewed {"data":{"subject":"201","count":4,"rating":5}, "head
```

## Using Multiple Channels

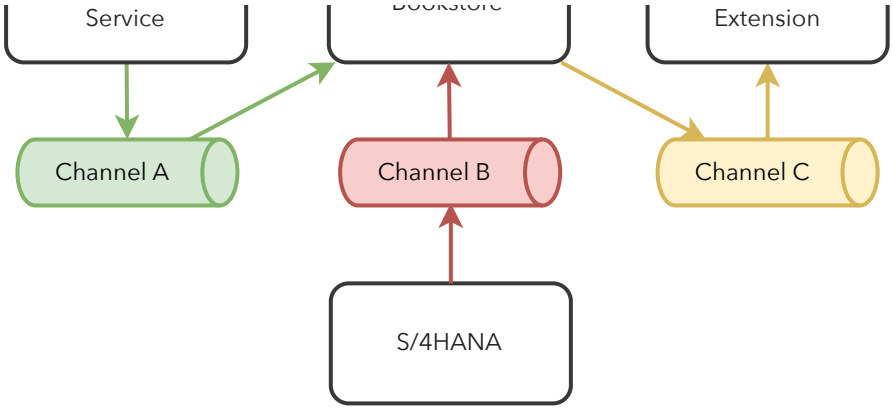
By default CAP uses a single message channel for all messages.

For example: If you consume messages from SAP S/4HANA in an enhanced version of `bookstore`, as well as emit messages a customer could subscribe and react to in a customer extension, the overall topology would look like that:




### Using Separate Channels

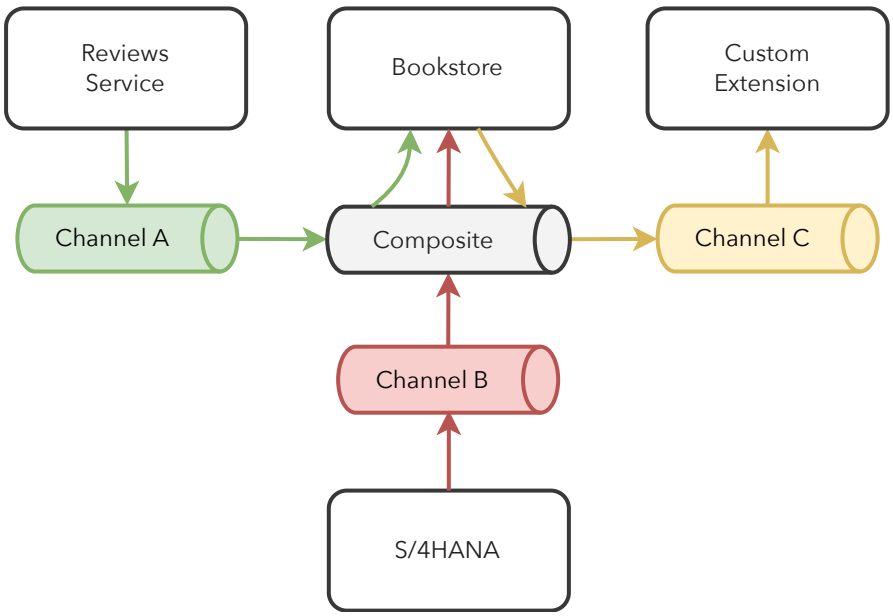
Now, sometimes you want to use separate channels for different emitters or receivers. Let's assume you want to have a dedicated channel for all events from SAP S/4HANA, and yet another separate one for all outgoing events, to which customer extensions can subscribe too. This situation is illustrated in this graphic:




This is possible when using low-level messaging, but comes at the price of loosing all advantages of conceptual-level messaging as explained in the following.

## Using `composite-messaging` Implementation

To avoid falling back to low-level messaging, CAP provides the `composite-messaging` implementation, which basically acts like a transparent dispatcher for both, inbound and outbound messages. The resulting topology would look like that:




> **Transparent Topologies**
>
> The `composite-messaging` implementation allows to flexibly change topologies of message channels at deployment time, without touching source code or models.

You would configure this in *bookstore* 's *package.json* as follows:

jsonc

```
"cds": {
  "requires": {
    "messaging": {
      "kind": "composite-messaging",
      "routes": {
        "ChannelA": ["**/ReviewsService/*"],
        "ChannelB": ["**/sap/s4/**"]
        "ChannelC": ["**/bookshop/**"]
      }
    },
    "ChannelA": {
      "kind": "enterprise-messaging", ...
    },
    "ChannelB": {
      "kind": "enterprise-messaging", ...
    },
    "ChannelC": {
      "kind": "enterprise-messaging", ...
    }
  }
}
```

In essence, you first configure a messaging service for each channel. In addition, you would configure the default *messaging* service to be of kind *composite-messaging* .

In the *routes* , you can use the glob pattern to define filters for event names, that means:

- ** will match any number of characters.
- * will match any number of characters except / and . .
- ? will match a single character.

> **TIP**
>
> You can also refer to events declared in CDS models, by using their fully qualified event name (unless annotation *@topic* is used on them).

## Low Level Messaging

In the previous sections it's documented how CAP promotes messaging on conceptual levels, staying agnostic to topologies and message brokers. While CAP strongly recommends staying on that level, CAP also offers lower-level messaging, which loses some of the advantages but still stays independent from specific message brokers.

> **Messaging as Just Another CAP Service**
>
> All messaging implementations are provided through class `cds.MessagingService` and broker-specific subclasses of that. This class is in turn a standard CAP service, derived from `cds.Service`, hence it's consumed as any other CAP service, and can also be extended by adding event handlers as usual.

### Configure Messaging Services

As with all other CAP services, add an entry to `cds.requires` in your *package.json* or *.cdsrc.json* like that:

```jsonc
"cds": {
  "requires": {
    "messaging": {
      "kind": // ...
    },
  }
}
```

↳ *Learn more about `cds.env` and `cds.requires`.*

You're free how you name your messaging service. Could be `messaging` as in the previous example, or any other name you choose. You can also configure multiple messages services with different names.

### Connect to the Messaging Service

Instead of connecting to an emitter service, connect to the messaging service:

```js
const messaging = await cds.connect.to('messaging')
```

### Emit Events to Messaging Service

```js
await messaging.emit ('ReviewsService.reviewed', { ... })
```

### Receive Events from Messaging Service

Instead of registering event handlers with a concrete emitter service, register handlers on the messaging service:

```js
messaging.on ('ReviewsService.reviewed', msg => console.log(msg))
```

### Declared Events and *@topic* Names

When declaring events in CDS models, be aware that the fully qualified name of the event is used as topic names when emitting to message brokers. Based on the following model, the resulting topic name is *my.namespace.SomeEventEmitter.SomeEvent* .

```cds
namespace my.namespace;
service SomeEventEmitter {
  event SomeEvent { ... }
}
```

If you want to manually define the topic, you can use the *@topic* annotation:

```cds
//...
@topic: 'some/very/different/topic-name'
event SomeEvent { ... }
```

### Conceptual vs. Low-Level Messaging

When looking at the previous code samples, you see that in contrast to conceptual messaging you need to provide fully qualified event names now. This is just one of the advantages you lose. Have a look at the following list of advantages you have using conceptual messaging and lose with low-level messaging.

- Service-local event names (as already mentioned)
- Event declarations (as they go with individual services)
- Generated typed API classes for declared events

> **Always prefer conceptual-level API over low-level API variants.**
>
> Besides the things listed above, this allows you to flexibly change topologies, such as starting with co-located services in a single process, and moving single services out to separate micro services later on.

## CloudEvents Standard

CAP messaging has built-in support for formatting event data compliant to the CloudEvents standard. Enable this using the `format` config option as follows:

```json
"cds": {
  "requires": {
    "messaging": {
      "format": "cloudevents"
    }
  }
}
```

With this setting, all mandatory and some more basic header fields, like `type`, `source`, `id`, `datacontenttype`, `specversion`, `time` are filled in automatically. The event name is used as `type`. The message payload is in the `data` property anyways.

> **CloudEvents is a wire protocol specification.**
>
> Application developers shouldn't have to care for such technical details. CAP ensures that for you, by filling in the respective fields behind the scenes.

## Using SAP Event Mesh

need to do is configuring CAP to use `enterprise-messaging`, usually in combination with `cloudevents` format, as in this excerpt from a *package.json*:

```jsonc
"cds": {
  "requires": {
    "messaging": {
      "[production]": {
        "kind": "enterprise-messaging",
        "format": "cloudevents"
      }
    }
  }
}
```

↳ *Learn more about `cds.env` profiles*

> **Read the guide**
>
> Find additional information about deploying SAP Event Mesh on SAP BTP in this guide: → ***Using SAP Event Mesh in BTP***

## Events from SAP S/4HANA

SAP S/4HANA integrates SAP Event Mesh for messaging. That makes it relatively easy for CAP-based applications to receive events from SAP S/4HANA systems.

In contrast to CAP, the asynchronous APIs of SAP S/4HANA are separate from the synchronous ones (OData, REST). So, the effort on the CAP side is to fill this gap. You can achieve it like that, for example, for an already imported SAP S/4HANA BusinessPartner API:

```cds
// filling in missing events as found on SAP Business Accelerator Hub
using { API_BUSINESS_PARTNER as S4 } from './API_BUSINESS_PARTNER';
extend service S4 with {
  event BusinessPartner.Created @(topic:'sap.s4.beh.businesspartner.v1.Busine
    BusinessPartner : String
  }
  event BusinessPartner.Changed @(topic:'sap.s4.beh.businesspartner.v1.Busine
```

```
  }
}
```

↳ *Learn more about importing SAP S/4HANA service APIs.*

With that gap filled, we can easily receive events from SAP S/4HANA the same way as from CAP services as explained in this guide, for example:

```js
const S4Bupa = await cds.connect.to ('API_BUSINESS_PARTNER')
S4Bupa.on ('BusinessPartner.Changed', msg => {...})
```

> **Read the guide**
>
> Find more detailed information specific to receiving events from SAP S/4HANA in this separate guide: → ***Receiving Events from SAP S/4HANA***

# Consuming Services

**This guide is available for Node.js and Java.**

Use the toggle in the title bar or press `v` to switch.

Table of Contents


- Introduction
  - Feature Overview
  - Tutorials and Examples
  - Define Scenario
- Get and Import an External Service API
  - From SAP Business Accelerator Hub
  - For a Remote CAP Service
  - Import API Definition
- Local Mocking
  - Add Mock Data
  - Run Local with Mocks
  - Mock Associations
  - Mock Remote Service as OData Service (Node.js)
- Execute Queries
  - Execute Queries with Node.js
  - Model Projections
  - Execute Queries on Projections to a Remote Service
  - Building Custom Requests with Node.js

    - Expose Remote Services
    - Expose Remote Services with Associations
    - Mashing up with Remote Services
    - Handle Mashups with Remote Services
    - Limitations and Feature Matrix
- Connect and Deploy
    - Using Destinations
    - Connect to Remote Services Locally
    - Connect to an Application Using the Same XSUAA (Forward Authorization Token)
    - Connect to an Application in Your Kyma Cluster
    - Deployment
    - Destinations and Multitenancy
- Add Qualities
    - Resilience
    - Tracing
- Feature Details
    - Legend
    - Supported Protocols
    - Querying API Features
    - Supported Projection Features
    - Supported Features for Application Defined Destinations


## Introduction

If you want to use data from other services or you want to split your application into multiple microservices, you need a connection between those services. We call them **remote services**. As everything in CAP is a service, remote services are modeled the same way as internal services – using CDS.

CAP supports service consumption with dedicated APIs to import service definitions, query remote services, mash up services, and work locally as much as possible.

For outbound remote service consumption, the following features are supported:

- OData V2
- OData V4
- Querying API
- Projections on remote services

## Tutorials and Examples

Most snippets in this guide are from the Build an Application End-to-End using CAP, Node.js, and VS Code tutorial, in particular Consume Remote Services from SAP S/4HANA Cloud Using CAP .

| Example | Description |
| --- | --- |
| Consume Remote Services from SAP S/4HANA Cloud Using CAP | End-to-end Tutorial, Node.js, SAP S/4HANA Cloud, SAP Business Accelerator Hub |
| Capire Bookshop (Fiori) | Example, Node.js, CAP-to-CAP |
| Example Application (Node.js) | Complete application from the end-to-end Tutorial |
| Example Application (Java) | Complete application from the end-to-end Tutorial |

## Define Scenario

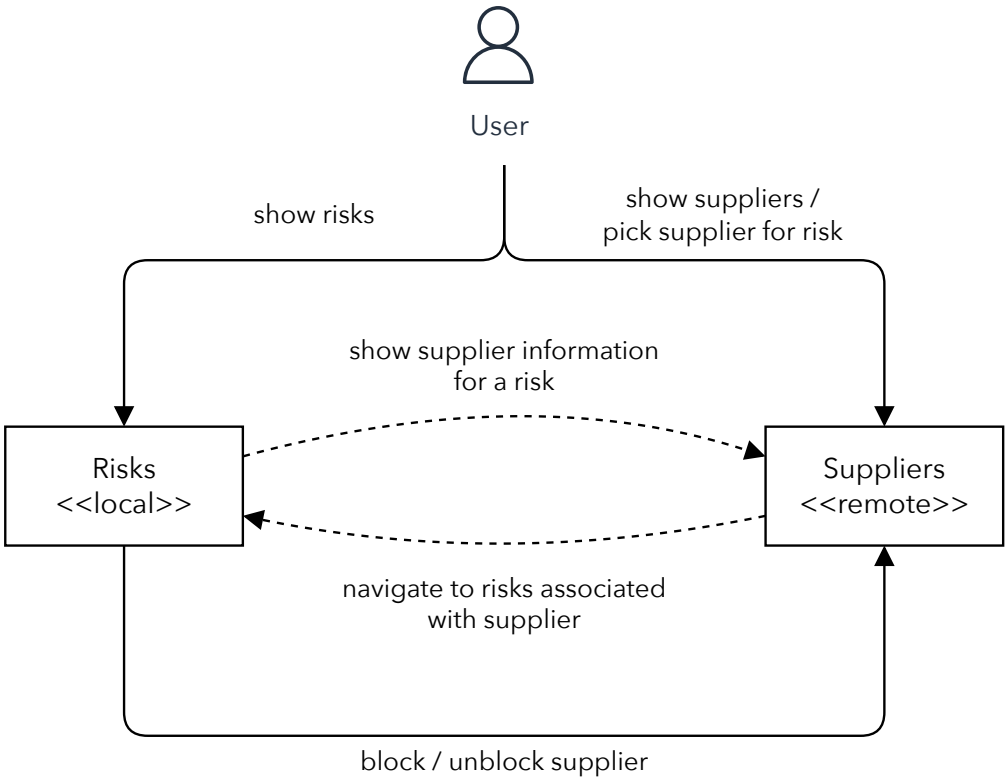
Before you start your implementation, you should define your scenario. Answering the following questions gets you started:

- What services (remote/CAP) are involved?
- How do they interact?
- What needs to be displayed on the UI?

You have all your answers and know your scenario, go on reading about external service APIs, getting an API definition from the SAP Business Accelerator Hub or from a CAP project, and importing an API definition to your project.

### Sample Scenario from End-to-End Tutorial

possible scenario:




> ***User Story***
>
> A company wants to ensure that goods are only sourced from suppliers with acceptable risks. There shall be a software system, that allows a clerk to maintain risks for suppliers and their mitigations. The system shall block the supplier used if risks can't be mitigated.

The application is an extension for SAP S/4HANA. It deals with *risks* and *mitigations* that are local entities in the application and *suppliers* that are stored in SAP S/4HANA Cloud. The application helps to reduce risks associated with suppliers by automatically blocking suppliers with a high risk using a remote API Call.

### Integrate

The user picks a supplier from the list. That list is coming from the remote system and is exposed by the CAP application. Then the user does a risk assessment. Additional supplier data, like name and blocked status, should be displayed on the UI as well, by integrating the remote supplier service into the local risk service.

### Extend

It should be also possible to search for suppliers and show the associated risks by extending the remote supplier service with the local risk service and its risks.

## Get and Import an External Service API

To communicate to remote services, CAP needs to know their definitions. Having the definitions in your project allows you to mock them during design time.

These definitions are usually made available by the service provider. As they aren't defined within your application but imported from outside, they're called *external* service APIs in CAP. Service APIs can be provided in different formats. Currently, *EDMX* files for OData V2 and V4 are supported.

### From SAP Business Accelerator Hub

The SAP Business Accelerator Hub provides many relevant APIs from SAP. You can download API specifications in different formats. If available, use the EDMX format. The EDMX format describes OData interfaces.

To download the Business Partner API (A2X) from SAP S/4HANA Cloud, go to section **API Resources**, select **API Specification**, and download the **EDMX** file.

↳ *Get more details in the end-to-end tutorial.*

### For a Remote CAP Service

We recommend using EDMX as exchange format. Export a service API to EDMX:

Mac/Linux | Windows | Powershell

```sh
cds compile srv -s OrdersService -2 edmx > OrdersService.edmx
```

↳ *You can try it with the orders sample in cap/samples.*

By default, CAP works with OData V4 and the EDMX export is in this protocol version as well. The `cds compile` command offers options for other OData versions and flavors, call `cds help compile` for more information.

> **Don't just copy the CDS file for a remote CAP service**
>
> Simply copying CDS files from a different application comes with the following issues:
>
> - The effective service API depends on the used protocol.

- CAP creates unneeded database tables and views for all entities in the file.

## Import API Definition

Import the API to your project using `cds import`.

```sh
cds import <input_file> --as cds
```

> `<input_file>` can be an EDMX (OData V2, OData V4), OpenAPI or AsyncAPI file.

| Option | Description |
| --- | --- |
| `--as cds` | The import creates a CDS file (for example *API_BUSINESS_PARTNER.cds*) instead of a CSN file. |

This adds the API in CDS format to the *srv/external* folder and also copies the input file into that folder.

Further, it adds the API as an external service to your *package.json*. You use this declaration later to connect to the remote service using a destination.

```json
"cds": {
    "requires": {
        "API_BUSINESS_PARTNER": {
            "kind": "odata-v2",
            "model": "srv/external/API_BUSINESS_PARTNER"
        }
    }
}
```

▶ *Options and flags in .cdsrc.json*

When importing the specification files, the `kind` is set according to the following mapping:

| Imported Format | Used `kind` |
| --- | --- |
| OData V2 | `odata-v2` |
| OData V4 | `odata` (alias for `odata-v4`) |

| | |
|---|---|
| OpenAPI | *rest* |
| AsyncAPI | *odata* |

↳ *Learn more about type mappings from OData to CDS and vice versa.*

> **TIP**
>
> Always use OData V4 ( *odata* ) when calling another CAP service.

## Local Mocking

When developing your application, you can mock the remote service.

### Add Mock Data

As for any other CAP service, you can add mocking data.

The CSV file needs to be added to the *srv/external/data* folder.

API_BUSINESS_PARTNER-A_BusinessPartner.csv

```csv
BusinessPartner;BusinessPartnerFullName;BusinessPartnerIsBlocked
1004155;Williams Electric Drives;false
1004161;Smith Batteries Ltd;false
1004100;Johnson Automotive Supplies;true
```

↳ *Find this source in the end-to-end tutorial*

↳ *Get more details in the end-to-end tutorial.*

### Run Local with Mocks

Start your project with the imported service definition.

The service is automatically mocked, as you can see in the log output on server start.

log

```
...

[cds] - model loaded from 8 file(s):

  ...
  ./srv/external/API_BUSINESS_PARTNER.cds
  ...

[cds] - connect using bindings from: { registry: '~/.cds-services.json' }
[cds] - connect to db > sqlite { database: ':memory:' }
 > filling sap.ui.riskmanagement.Mitigations from ./db/data/sap.ui.riskmanage
 > filling sap.ui.riskmanagement.Risks from ./db/data/sap.ui.riskmanagement-R
 > filling API_BUSINESS_PARTNER.A_BusinessPartner from ./srv/external/data/AP
/> successfully deployed to sqlite in-memory db

[cds] - serving RiskService { at: '/service/risk', impl: './srv/risk-service.
[cds] - mocking API_BUSINESS_PARTNER { at: '/api-business-partner' }

[cds] - launched in: 1.104s
[cds] - server listening on { url: 'http://localhost:4004' }
[ terminate with ^C ]
```

## Mock Associations

You can't get data from associations of a mocked service out of the box.

The associations of imported services lack information how to look up the associated records. This missing relation is expressed with an empty key definition at the end of the association declaration in the CDS model ( `{ }` ).

srv/external/API_BUSINESS_PARTNER.cds

cds

```
entity API_BUSINESS_PARTNER.A_BusinessPartner {
  key BusinessPartner : LargeString;
  BusinessPartnerFullName : LargeString;
```

```
  ...

  to_BusinessPartnerAddress :
    Association to many API_BUSINESS_PARTNER.A_BusinessPartnerAddress {  };
};

entity API_BUSINESS_PARTNER.A_BusinessPartnerAddress {
  key BusinessPartner : String(10);
  key AddressID : String(10);

  ...
};
```

To mock an association, you have to modify the imported file. Before doing any modifications, create a local copy and add it to your source code management system.

```sh
cp srv/external/API_BUSINESS_PARTNER.cds srv/external/API_BUSINESS_PARTNER-or
git add srv/external/API_BUSINESS_PARTNER-orig.cds
...
```

Import the CDS file again, just using a different name:

```sh
cds import ~/Downloads/API_BUSINESS_PARTNER.edmx --keep-namespace \
    --as cds --out srv/external/API_BUSINESS_PARTNER-new.cds
```

Add an *on* condition to express the relation:

srv/external/API_BUSINESS_PARTNER-new.cds

```cds
entity API_BUSINESS_PARTNER.A_BusinessPartner {
  // ...
  to_BusinessPartnerAddress :
      Association to many API_BUSINESS_PARTNER.A_BusinessPartnerAddress
      on to_BusinessPartnerAddress.BusinessPartner = BusinessPartner;
};
```

Don't add any keys or remove empty keys, which would change it to a managed association. Added fields aren't known in the service and lead to runtime errors.

unmodified file with the newly imported file:

```sh
git merge-file API_BUSINESS_PARTNER.cds \
               API_BUSINESS_PARTNER-orig.cds \
               API_BUSINESS_PARTNER-new.cds
mv API_BUSINESS_PARTNER-new.cds API_BUSINESS_PARTNER-orig.cds
```

To prevent accidental loss of modifications, the *cds import --as cds* command refuses to overwrite modified files based on a "checksum" that is included in the file.

## Mock Remote Service as OData Service (Node.js)

As shown previously you can run one process including a mocked external service. However, this mock doesn't behave like a real external service. The communication happens in-process and doesn't use HTTP or OData. For a more realistic testing, let the mocked service run in a separate process.

First install the required packages:

```sh
npm add @sap-cloud-sdk/http-client@3.x @sap-cloud-sdk/connectivity@3.x @sap-c
```

Then start the CAP application with the mocked remote service only:

```sh
cds mock API_BUSINESS_PARTNER
```

If the startup is completed, run *cds watch* in the same project from a **different** terminal:

```sh
cds watch
```

CAP tracks locally running services. The mocked service *API_BUSINESS_PARTNER* is registered in file *~/.cds-services.json*. *cds watch* searches for running services in that file and connects to them.

Node.js only supports *OData V4* protocol and so does the mocked service. There might still be some differences to the real remote service if it uses a different protocol, but it's much closer to it than using only one instance. In the console output, you can also easily see how the communication between the two processes happens.

## Execute Queries

You can send requests to remote services using CAP's powerful querying API.

### Execute Queries with Node.js

Connect to the service before sending a request, as usual in CAP:

```js
const bupa = await cds.connect.to('API_BUSINESS_PARTNER');
```

Then execute your queries using the Querying API:

```js
const { A_BusinessPartner } = bupa.entities;
const result = await bupa.run(SELECT(A_BusinessPartner).limit(100));
```

We recommend limiting the result set and avoid the download of large data sets in a single request. You can *`limit`* the result as in the example: *`.limit(100)`* .

Many features of the querying API are supported for OData services. For example, you can resolve associations like this:

```js
const { A_BusinessPartner } = bupa.entities;
const result = await bupa.run(SELECT.from(A_BusinessPartner, bp => {
    bp('BusinessPartner'),
    bp.to_BusinessPartnerAddress(addresses => {
      addresses('*')
    })
  }).limit(100));
```

↳ *Learn more about querying API examples.*

↳ *Learn more about supported querying API features.*

### Model Projections

External service definitions, like generated CDS or CSN files during import, can be used as any other CDS definition, but they **don't** generate database tables and views unless they are mocked.

fields in a projection in your namespace. Your implementation is then independent of the remote service implementation and you request only the information that you require.

```cds
using {  API_BUSINESS_PARTNER as bupa } from '../srv/external/API_BUSINESS_PA

entity Suppliers as projection on bupa.A_BusinessPartner {
  key BusinessPartner as ID,
  BusinessPartnerFullName as fullName,
  BusinessPartnerIsBlocked as isBlocked,
}
```

As the example shows, you can use field aliases as well.

↳ *Learn more about supported features for projections.*

## Execute Queries on Projections to a Remote Service

Connect to the service before sending a request, as usual in CAP:

```js
const bupa = await cds.connect.to('API_BUSINESS_PARTNER');
```

Then execute your queries:

```js
const suppliers = await bupa.run(SELECT(Suppliers).where({ID}));
```

CAP resolves projections and does the required mapping, similar to databases.

A brief explanation, based on the previous query, what CAP does:

- Resolves the *`Suppliers`* projection to the external service interface *`API_BUSINESS_PARTNER.A_Business_Partner`*.
- The **where** condition for field *`ID`* will be mapped to the *`BusinessPartner`* field of *`A_BusinessPartner`*.
- The result is mapped back to the *`Suppliers`* projection, so that values for the *`BusinessPartner`* field are mapped back to *`ID`*.

This makes it convenient to work with external services.

If you can't use the querying API, you can craft your own HTTP requests using *send*:

```js
bupa.send({
  method: 'PATCH',
  path: A_BusinessPartner,
  data: {
    BusinessPartner: 1004155,
    BusinessPartnerIsBlocked: true
  }
})
```

↳ *Learn more about the send API.*

## Integrate and Extend

By creating projections on remote service entities and using associations, you can create services that combine data from your local service and remote services.

What you need to do depends on the scenarios and how your remote services should be integrated into, as well as extended by your local services.

### Expose Remote Services

To expose a remote service entity, you add a projection on it to your CAP service:

```cds
using {  API_BUSINESS_PARTNER as bupa } from '../srv/external/API_BUSINESS_PA

extend service RiskService with {
  entity BusinessPartners as projection on bupa.A_BusinessPartner;
}
```

CAP automatically tries to delegate queries to database entities, which don't exist as you're pointing to an external service. That behavior would produce an error like this:

```
<code>500</code>
<message>SQLITE_ERROR: no such table: RiskService_BusinessPartners in: SELECT
</error>
```

To avoid this error, you need to handle projections. Write a handler function to delegate a query to the remote service and run the incoming query on the external service.

Node.js Java

```js
module.exports = cds.service.impl(async function() {
  const bupa = await cds.connect.to('API_BUSINESS_PARTNER');

  this.on('READ', 'BusinessPartners', req => {
      return bupa.run(req.query);
  });
});
```

↳ *For Node.js, get more details in the end-to-end tutorial.*

> **WARNING**
>
> If you receive `404` errors, check if the request contains fields that don't exist in the service and start with the name of an association. `cds import` adds an empty keys declaration ( `{ }` ) to each association. Without this declaration, foreign keys for associations are generated in the runtime model, that don't exist in the real service. To solve this problem, you need to reimport the external service definition using `cds import` .

This works when accessing the entity directly. Additional work is required to support navigation and expands from or to a remote entity.

Instead of exposing the remote service's entity unchanged, you can model your own projection. For example, you can define a subset of fields and change their names.

> **TIP**
>
> CAP does the magic that maps the incoming query, according to your projections, to the remote service and maps back the result.

```cds
using { API_BUSINESS_PARTNER as bupa } from '../srv/external/API_BUSINESS_PAR
```

```
  entity Suppliers as projection on bupa.A_BusinessPartner {
    key BusinessPartner as ID,
    BusinessPartnerFullName as fullName,
    BusinessPartnerIsBlocked as isBlocked
  }
}
```

```js
module.exports = cds.service.impl(async function() {
  const bupa = await cds.connect.to('API_BUSINESS_PARTNER');

  this.on('READ', 'Suppliers', req => {
      return bupa.run(req.query);
  });
});
```

↳ *Learn more about queries on projections to remote services.*

## Expose Remote Services with Associations

It's possible to expose associations of a remote service entity. You can adjust the projection for the association target and change the name of the association:

```cds
using { API_BUSINESS_PARTNER as bupa } from '../srv/external/API_BUSINESS_PAR

extend service RiskService with {
  entity Suppliers as projection on bupa.A_BusinessPartner {
    key BusinessPartner as ID,
    BusinessPartnerFullName as fullName,
    BusinessPartnerIsBlocked as isBlocked,
    to_BusinessPartnerAddress as addresses: redirected to SupplierAddresses
  }

  entity SupplierAddresses as projection on bupa.A_BusinessPartnerAddress {
    BusinessPartner as bupaID,
    AddressID as ID,
    CityName as city,
    StreetName as street,
    County as county
```

```
}
```

As long as the association is only resolved using expands (for example *`.../risk/Suppliers?$expand=addresses`* ), a handler for the **source entity** is sufficient:

```js
this.on('READ', 'Suppliers', req => {
    return bupa.run(req.query);
});
```

If you need to resolve the association using navigation or request it independently from the source entity, add a handler for the **target entity** as well:

```js
this.on('READ', 'SupplierAddresses', req => {
    return bupa.run(req.query);
});
```

As usual, you can put two handlers into one handler matching both entities:

```js
this.on('READ', ['Suppliers', 'SupplierAddresses'], req => {
    return bupa.run(req.query);
});
```

## Mashing up with Remote Services

You can combine local and remote services using associations. These associations need manual handling, because of their different data sources.

### Integrate Remote into Local Services

Use managed associations from local entities to remote entities:

```cds
@path: 'service/risk'
service RiskService {
  entity Risks : managed {
    key ID       : UUID  @(Core.Computed : true);
    title        : String(100);
    prio         : String(5);
    supplier     : Association to Suppliers;
```

```
  entity Suppliers as projection on BusinessPartner.A_BusinessPartner {
    key BusinessPartner as ID,
    BusinessPartnerFullName as fullName,
    BusinessPartnerIsBlocked as isBlocked,
  };
}
```

### Extend a Remote by a Local Service

You can augment a projection with a new association, if the required fields for the on condition are present in the remote service. The use of managed associations isn't possible, because this requires to create new fields in the remote service.

cds

```
entity Suppliers as projection on bupa.A_BusinessPartner {
  key BusinessPartner as ID,
  BusinessPartnerFullName as fullName,
  BusinessPartnerIsBlocked as isBlocked,
  risks : Association to many Risks on risks.supplier.ID = ID,
};
```

## Handle Mashups with Remote Services

Depending on how the service is accessed, you need to support direct requests, navigation, or expands. CAP resolves those three request types only for service entities that are served from the database. When crossing the boundary between database and remote sourced entities, you need to take care of those requests.

The list of required implementations for mashups explains the different combinations.

### Handle Expands Across Local and Remote Entities

Expands add data from associated entities to the response. For example, for a risk, you want to display the suppliers name instead of just the technical ID. But this property is part of the (remote) supplier and not part of the (local) risk.

↳ *Get more details in the end-to-end tutorial.*

To handle expands, you need to add a handler for the main entity:

1. Check if a relevant `$expand` column is present.

3. Get the data for the request.
4. Execute a new request for the expand.
5. Add the expand data to the returned data from the request.

Example of a CQN request with an expand:

```json
{
  "from": { "ref": [ "RiskService.Suppliers" ] },
  "columns": [
    { "ref": [ "ID" ] },
    { "ref": [ "fullName" ] },
    { "ref": [ "isBlocked" ] },
    { "ref": [ "risks" ] },
    { "expand": [
      { "ref": [ "ID" ] },
      { "ref": [ "title" ] },
      { "ref": [ "descr" ] },
      { "ref": [ "supplier_ID" ] }
    ] }
  ]
}
```

↳ *See an example how to handle expands in Node.js.*

Expands across local and remote can cause stability and performance issues. For a list of items, you need to collect all IDs and send it to the database or the remote system. This can become long and may exceed the limits of a URL string in case of OData. Do you really need expands for a list of items?

```http
GET /service/risk/Risks?$expand=supplier
```

Or is it sufficient for single items?

```http
GET /service/risk/Risks(545A3CF9-84CF-46C8-93DC-E29F0F2BC6BE)/?$expand=suppli
```

**Keep performance in mind**

Consider to reject expands if it's requested on a list of items.

Navigations allow to address items via an association from a different entity:

```http
GET /service/risks/Risks(20466922-7d57-4e76-b14c-e53fd97dcb11)/supplier
```

The CQN consists of a *from* condition with 2 values for *ref*. The first *ref* selects the record of the source entity of the navigation. The second *ref* selects the name of the association, to navigate to the target entity.

```json
{
  "from": {
    "ref": [ {
      "id": "RiskService.Risks",
      "where": [
        { "ref": [ "ID" ] },
        "=",
        { "val": "20466922-7d57-4e76-b14c-e53fd97dcb11" }
      ]},
      "supplier"
    ]
  },
  "columns": [
    { "ref": [ "ID" ] },
    { "ref": [ "fullName" ] },
    { "ref": [ "isBlocked" ] }
  ],
  "one": true
}
```

To handle navigations, you need to check in your code if the *from.ref* object contains 2 elements. Be aware, that for navigations the handler of the **target** entity is called.

If the association's on condition equals the key of the source entity, you can directly select the target entity using the key's value. You find the value in the *where* block of the first *from.ref* entry.

Otherwise, you need to select the source item using that *where* block and take the required fields for the associations on condition from that result.

↳ *See an example how to handle navigations in Node.js.*

## Limitations and Feature Matrix

You need additional logic, if remote entities are in the game. The following table shows what is required. "Local" is a database entity or a projection on a database entity.

| Request | Example | Implementation |
| --- | --- | --- |
| Local (including navigations and expands) | `/service/risks/Risks` | Handled by CAP |
| Local: Expand remote | `/service/risks/Risks?$expand=supplier` | Delegate query w/o expand to local service and implement expand. |
| Local: Navigate to remote | `/service/risks(...)/supplier` | Implement navigation and delegate query target to remote service. |
| Remote (including navigations and expands to the same remote service) | `/service/risks/Suppliers` | Delegate query to remote service |
| Remote: Expand local | `/service/risks/Suppliers?$expand=risks` | Delegate query w/o expand to remote service and implement expand. |
| Remote: Navigate to local | `/service/Suppliers(...)/risks` | Implement navigation, delegate query for target to local service |

### Transient Access vs. Replication

> This chapter shows only techniques for transient access.

The following matrix can help you to find the best approach for your scenario:

| Feature | Transient Access | Replication |
| --- | --- | --- |
| Filtering on local **or** remote fields [1] | Possible | Possible |
| Filtering on local **and** remote fields [2] | Not possible | Possible |
| Relationship: Uni-/Bidirectional associations | Possible | Possible |

| | | |
|---|---|---|
| Relationship: Flatten | Not possible | Possible |
| Evaluate user permissions in remote system | Possible | Requires workarounds [3] |
| Data freshness | Live data | Outdated until replicated |
| Performance | Degraded [4] | Best |

> [1] It's **not required** to filter both, on local and remote fields, in the same request.
>
> [2] It's **required** to filter both, on local and remote fields, in the same request.
>
> [3] Because replicated data is accessed, the user permission checks of the remote system aren't evaluated.
>
> [4] Depends on the connectivity and performance of the remote system.

# Connect and Deploy

## Using Destinations

Destinations contain the necessary information to connect to a remote system. They're basically an advanced URL, that can carry additional metadata like, for example, the authentication information.

You can choose to use SAP BTP destinations or application defined destinations.

### Use SAP BTP Destinations

CAP leverages the destination capabilities of the SAP Cloud SDK.

#### Create Destinations on SAP BTP

Create a destination using one or more of the following options.

- **Register a system in your global account:** You can check here how to Register an SAP System in your SAP BTP global account and which systems are supported for registration. Once the system is registered and assigned to your subaccount, you can

instance.

- **Connect to an on-premise system:** With SAP BTP Cloud Connector , you can create a connection from your cloud application to an on-premise system.
- **Manually create destinations:** You can create destinations manually in your SAP BTP subaccount. See section destinations in the SAP BTP documentation.
- **Create a destination to your application:** If you need a destination to your application, for example, to call it from a different application, then you can automatically create it in the MTA deployment.

**Use Destinations with Node.js**

In your *package.json*, a configuration for the `API_BUSINESS_PARTNER` looks like this:

```json
"cds": {
  "requires": {
    "API_BUSINESS_PARTNER": {
      "kind": "odata",
      "model": "srv/external/API_BUSINESS_PARTNER"
    }
  }
}
```

If you've imported the external service definition using `cds import` , an entry for the service in the *package.json* has been created already. Here you specify the name of the destination in the `credentials` block.

In many cases, you also need to specify the `path` prefix to the service, which is added to the destination's URL. For services listed on the SAP Business Accelerator Hub, you can find the path in the linked service documentation.

Since you don't want to use the destination for local testing, but only for production, you can profile it by wrapping it into a `[production]` block:

```json
"cds": {
  "requires": {
    "API_BUSINESS_PARTNER": {
      "kind": "odata",
      "model": "srv/external/API_BUSINESS_PARTNER",
      "[production]": {
        "credentials": {
          "destination": "S4HANA",
```

```
        }
      }
    }
  }
}
```

Additionally, you can provide destination options inside a *destinationOptions* object:

jsonc

```
"cds": {
  "requires": {
    "API_BUSINESS_PARTNER": {
      /* ... */
      "[production]": {
        "credentials": {
          /* ... */
        },
        "destinationOptions": {
          "selectionStrategy": "alwaysSubscriber",
          "useCache": true
        }
      }
    }
  }
}
```

The *selectionStrategy* property controls how a destination is resolved.

The *useCache* option controls whether the SAP Cloud SDK caches the destination. It's enabled by default but can be disabled by explicitly setting it to *false* . Read Destination Cache to learn more about how the cache works.

If you want to configure additional headers for the HTTP request to the system behind the destination, for example an Application Interface Register (AIR) header, you can specify such headers in the destination definition itself using the property *URL.headers.<header-key>* .

### Use Application Defined Destinations

If you don't want to use SAP BTP destinations, you can also define destinations, which means the URL, authentication type, and additional configuration properties, in your application configuration or code.

Application defined destinations support a subset of properties and authentication types of the SAP BTP destination service.

You specify the destination properties in *`credentials`* instead of putting the name of a destination there.

This is an example of a destination using basic authentication:

```jsonc
"cds": {
  "requires": {
    "REVIEWS": {
      "kind": "odata",
      "model": "srv/external/REVIEWS",
      "[production]": {
        "credentials": {
          "url": "https://reviews.ondemand.com/reviews",
          "authentication": "BasicAuthentication",
          "username": "<set from code or env>",
          "password": "<set from code or env>",
          "headers": {
            "my-header": "header value"
          },
          "queries": {
            "my-url-param": "url param value"
          }
        }
      }
    }
  }
}
```

↳ *Supported destination properties.*

> **WARNING**
>
> You shouldn't put any sensitive information here.

Instead, set the properties in the bootstrap code of your CAP application:

```js
const cds = require("@sap/cds");

if (cds.env.requires?.credentials?.authentication === "BasicAuthentication")
  const credentials = /* read your credentials */
  cds.env.requires.credentials.username = credentials.username;
```

```
}
```

You might also want to set some values in the application deployment. This can be done using env variables. For this example, the env variable for the URL would be `cds_requires_REVIEWS_credentials_destination_url` .

This variable can be parameterized in the *manifest.yml* for a `cf push` based deployment:

manifest.yml

```yaml
applications:
- name: reviews
  ...
  env:
    cds_requires_REVIEWS_credentials_url: ((reviews_url))
```

```sh
cf push --var reviews_url=https://reviews.ondemand.com/reviews
```

The same can be done using *mtaext* file for MTA deployment.

If the URL of the target service is also part of the MTA deployment, you can automatically receive it as shown in this example:

mta.yaml

```yaml
 - name: reviews
   provides:
    - name: reviews-api
      properties:
        reviews-url: ${default-url}
 - name: bookshop
   requires:
    ...
    - name: reviews-api
   properties:
     cds_requires_REVIEWS_credentials_url: ~{reviews-api/reviews-url}
```

.env

> **WARNING**
>
> For the *configuration path*, you **must** use the underscore (" `_` ") character as delimiter. CAP supports dot (" `.` ") as well, but Cloud Foundry won't recognize variables using dots. Your *service name* **mustn't** contain underscores.

### Implement Application Defined Destinations in Node.js

There is no API to create a destination in Node.js programmatically. However, you can change the properties of a remote service before connecting to it, as shown in the previous example.

## Connect to Remote Services Locally

If you use SAP BTP destinations, you can access them locally using CAP's hybrid testing capabilities with the following procedure:

### Bind to Remote Destinations

Your local application needs access to an XSUAA and Destination service instance in the same subaccount where the destination is:

1. Login to your Cloud Foundry org and space
2. Create an XSUAA service instance and service key:

```sh
cf create-service xsuaa application cpapp-xsuaa
cf create-service-key cpapp-xsuaa cpapp-xsuaa-key
```

3. Create a Destination service instance and service key:

```sh
cf create-service destination lite cpapp-destination
cf create-service-key cpapp-destination cpapp-destination-key
```

4. Bind to XSUAA and Destination service:

```sh
cds bind -2 cpapp-xsuaa,cpapp-destination
```

↳ *Learn more about `cds bind`.*

Add the destination for the remote service to the `hybrid` profile in the *.cdsrc-private.json* file:

```jsonc
{
  "requires": {
    "[hybrid]": {
      "auth": {
        /* ... */
      },
      "destinations": {
        /* ... */
      },
      "API_BUSINESS_PARTNER": {
        "credentials": {
          "path": "/sap/opu/odata/sap/API_BUSINESS_PARTNER",
          "destination": "cpapp-bupa"
        }
      }
    }
  }
}
```

Run your application with the Destination service:

```sh
cds watch --profile hybrid
```

> **TIP**
>
> If you are developing in the Business Application Studio and want to connect to an on-premise system, you will need to do so via Business Application Studio's built-in proxy, for which you need to add configuration in an `.env` file. See Connecting to External Systems From the Business Application Studio for more details.

## Connect to an Application Using the Same XSUAA (Forward Authorization Token)

If your application consists of microservices and you use one (or more) as a remote service as described in this guide, you can leverage the same XSUAA instance. In that case, you don't need an SAP BTP destination at all.

authorization token. The URL of the remote service can be injected into the application in the MTA or Cloud Foundry deployment using application defined destinations.

### Forward Authorization Token with Node.js

To enable the token forwarding, set the `forwardAuthToken` option to `true` in your application defined destination:

```json
{
  "requires": {
    "kind": "odata",
    "model": "./srv/external/OrdersService",
    "credentials": {
      "url": "<set via env var in deployment>",
      "forwardAuthToken": true
    }
  }
}
```

## Connect to an Application in Your Kyma Cluster

The Istio service mesh provides secure communication between the services in your service mesh. You can access a service in your applications' namespace by just reaching out to `http://<service-name>` or using the full hostname `http://<service-name>.<namespace>.svc.cluster.local`. Istio sends the requests through an mTLS tunnel.

With Istio, you can further secure the communication by configuring authentication and authorization for your services

## Deployment

Your micro service needs bindings to the **XSUAA** and **Destination** service to access destinations on SAP BTP. If you want to access an on-premise service using **Cloud Connector**, then you need a binding to the **Connectivity** service as well.

↳ *Learn more about deploying CAP applications.*
↳ *Learn more about deploying an application using the end-to-end tutorial.*

### Add Required Services to MTA Deployments

guide and make some additional adjustments to the generated *mta.yml* file.

```sh
cds add xsuaa,destination,connectivity --for production
```

▶ *Learn what this does in the background...*

Build your application:

```sh
mbt build -t gen --mtar mta.tar
```

Now you can deploy it to Cloud Foundry:

```sh
cf deploy gen/mta.tar
```

### Connectivity Service Credentials on Kyma

The secret of the connectivity service on Kyma needs to be modified for the Cloud SDK to connect to on-premise destinations.

↳ *Support for Connectivity Service Secret in Node.js*

## Destinations and Multitenancy

With the destination service, you can access destinations in your provider account, the account your application is running in, and destinations in the subscriber accounts of your multitenant-aware application.

### Use Destinations from Subscriber Account

Customers want to see business partners from, for example, their SAP S/4 HANA system.

As provider, you need to define a name for a destination, which enables access to systems of the subscriber of your application. In addition, your multitenant application or service needs to have a dependency to the destination service. For destinations in an on-premise system, the connectivity service must be bound.

The subscriber needs to create a destination with that name in their subscriber account, for example, pointing to their SAP S/4HANA system.

The destination is read from the tenant of the request's JWT (authorization) token. If no JWT token is present *or the destination isn't found*, the destination is read from the tenant of the application's XSUAA binding.

> **JWT token vs. XSUAA binding**
>
> Using the tenant of the request's JWT token means reading from the **subscriber subaccount** for a multitenant application. The tenant of the application's XSUAA binding points to the destination of the **provider subaccount**, the account where the application is deployed to.

You can change the destination lookup behavior as follows:

jsonc

```
"cds": {
  "requires": {
    "SERVICE_FOR_PROVIDER": {
      /* ... */
      "credentials": {
        /* ... */
      },
      "destinationOptions": {
        "selectionStrategy": "alwaysProvider",
        "jwt": null
      }
    }
  }
}
```

Setting the `selectionStrategy` property for the destination options to `alwaysProvider`, you can ensure that the destination is always read from your provider subaccount. With that you ensure that a subscriber cannot overwrite your destination.

Set the destination option `jwt` to `null`, if you don't want to pass the request's JWT to SAP Cloud SDK. Passing the request's JWT to SAP Cloud SDK has implications on, amongst others, the effective defaults for selection strategy and isolation level. In rare cases, these defaults are not suitable, for example when the request to the upstream server does not depend on the current user. Please see Authentication and JSON Web Token (JWT) Retrieval for more details.

## Resilience

There are two ways to make your outbound communications resilient:

1. Run your application in a service mesh (for example, Istio, Linkerd, etc.). For example, Kyma is provided as service mesh.
2. Implement resilience in your application.

Refer to the documentation for the service mesh of your choice for instructions. No code changes should be required.

To build resilience into your application, there are libraries to help you implement functions, like doing retries, circuit breakers or implementing fallbacks.

There's no resilience library provided out of the box for CAP Node.js. However, you can use packages provided by the Node.js community. Usually, they provide a function to wrap your code that adds the resilience logic.

### Resilience in Kyma

Kyma clusters run an Istio service mesh. Istio allows to configure resilience for the network destinations of your service mesh.

## Tracing

CAP adds headers for request correlation to its outbound requests that allows logging and tracing across micro services.

↳ *Learn more about request correlation in Node.js.*

---

# Feature Details

## Legend

| | |
|---|---|
| ✓ | supported |
| ✗ | not supported |

## Supported Protocols

| Protocol | Java | Node.js |
|---|---|---|
| odata-v2 | ✓ | ✓ |
| odata-v4 | ✓ | ✓ |
| rest | ✗ | ✓ |

> **TIP**
>
> The Node.js runtime supports `odata` as an alias for `odata-v4` as well.

## Querying API Features

| Feature | Java | Node.js |
|---|---|---|
| READ | ✓ | ✓ |
| INSERT/UPDATE/DELETE | ✓ | ✓ |
| Actions | ✓ | ✓ |
| `columns` | ✓ | ✓ |
| `where` | ✓ | ✓ |
| `orderby` | ✓ | ✓ |
| `limit` (top & skip) | ✓ | ✓ |
| `$apply` (groupedby, ...) | ✗ | ✗ |
| `$search` (OData v4) | ✓ | ✓ |

| | | |
|---|---|---|
| `search` (SAP OData v2 extension) | ✓ | ✓ |

## Supported Projection Features

| Feature | Java | Node.js |
|---|---|---|
| Resolve projections to remote services | ✓ | ✓ |
| Resolve multiple levels of projections to remote services | ✓ | ✓ |
| Aliases for fields | ✓ | ✓ |
| `excluding` | ✓ | ✓ |
| Resolve associations (within the same remote service) | ✓ | ✓ |
| Redirected associations | ✓ | ✓ |
| Flatten associations | ✗ | ✗ |
| `where` conditions | ✗ | ✗ |
| `order by` | ✗ | ✗ |
| Infix filter for associations | ✗ | ✗ |
| Model Associations with mixins | ✓ | ✓ |

## Supported Features for Application Defined Destinations

The following properties and authentication types are supported for *application defined destinations*:

### Properties

These destination properties are fully supported by both, the Java and the Node.js runtime.

> **TIP**
>
> This list specifies the properties for application defined destinations.

| | |
|---|---|
| *url* | |
| *authentication* | Authentication type |
| *username* | User name for BasicAuthentication |
| *password* | Password for BasicAuthentication |
| *headers* | Map of HTTP headers |
| *queries* | Map of URL parameters |
| *forwardAuthToken* | Forward auth token |

↳ *Destination Type in SAP Cloud SDK for JavaScript*

## Authentication Types

| Authentication Types | Java | Node.js |
|---|---|---|
| NoAuthentication | ✓ | ✓ |
| BasicAuthentication | ✓ | ✓ |
| TokenForwarding | ✓ | ✗ Use *forwardAuthToken* |
| OAuth2ClientCredentials | code only | ✗ |
| UserTokenAuthentication | code only | ✗ |

# 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
  - Service-Centric Paradigm
  - Ubiquitous Events
  - Event Handlers
- Service Definitions
  - Services as APIs
  - Services as Facades
  - Denormalized Views
  - Auto-Exposed Entities
  - Redirected Associations
- Generic Providers
  - Serving CRUD Requests
  - Deep Reads / Writes
  - Auto-Generated Keys
  - Searching Data
  - Pagination & Sorting
  - Concurrency Control
- Input Validation
  - @readonly

    - @assert.unique
    - @assert.target
    - @assert.format
    - @assert.range
    - @assert.notNull
- Custom Logic
    - Custom Event Handlers
    - Hooks: on, before, after
    - Within Event Handlers
- Actions & Functions
    - Implementing Actions / Functions
    - Calling Actions / Functions
- Serving Media Data
    - Annotating Media Elements
    - Reading Media Resources
    - Creating a Media Resource
    - Updating Media Resources
    - Deleting Media Resources
    - Using External Resources
    - Conventions & Limitations
- Best Practice: Single-Purposed Services
- Best Practice: Late-Cut Microservices

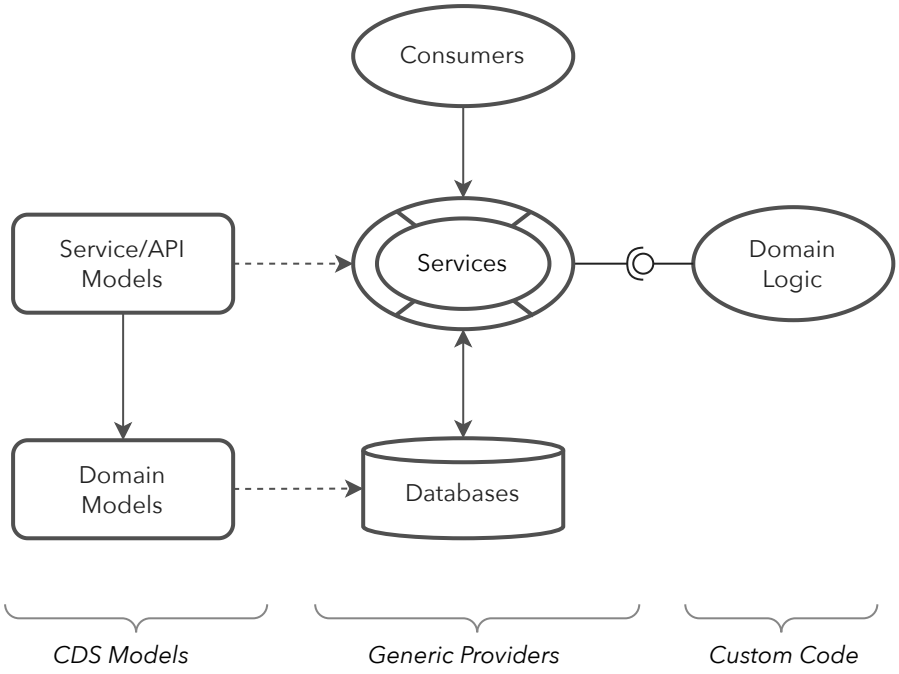

---

## Intro: Core Concepts

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

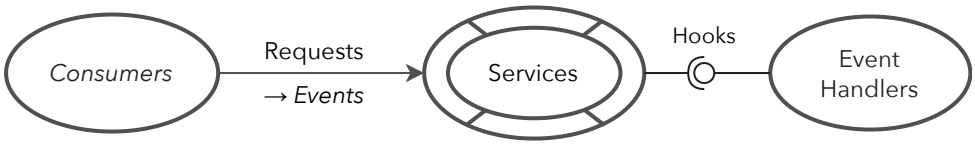
### Service-Centric Paradigm

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




## 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.

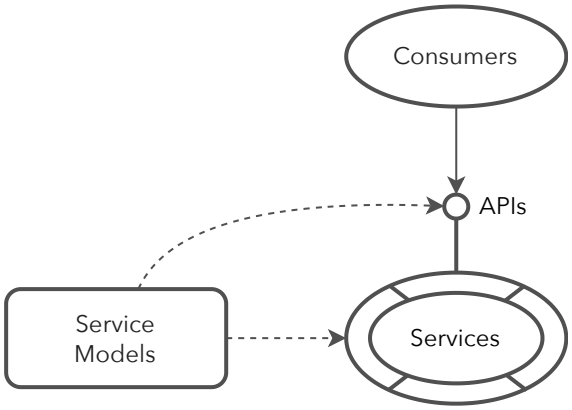
# Service Definitions

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`* .



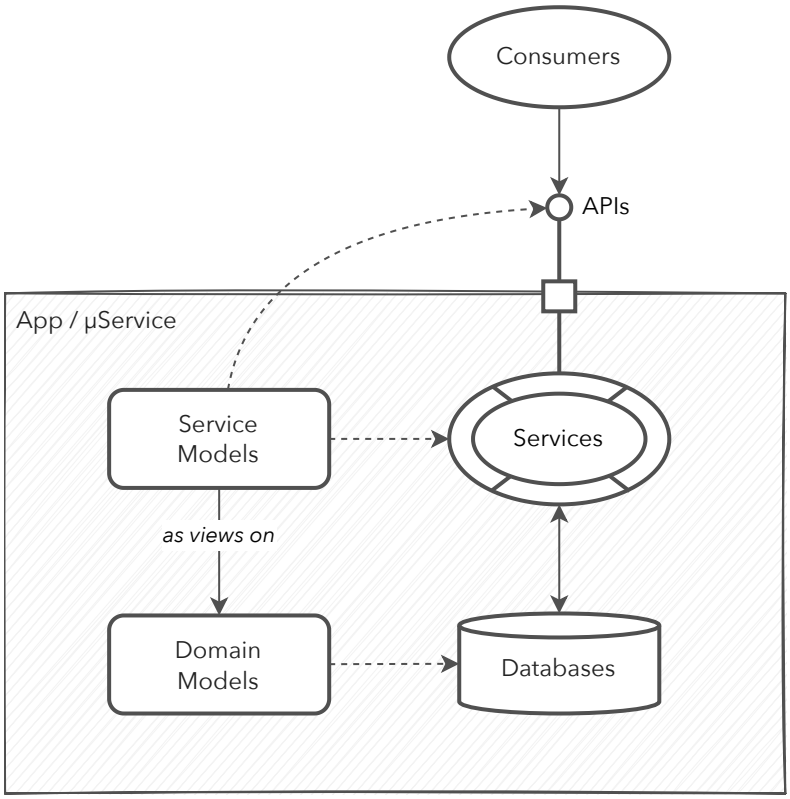

Simple service definitions like that are all we need to run full-fledged servers out of the box, served by CAP's generic runtimes, without any implementation coding required.

## Services as Facades

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

```
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 */
```

```
  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 **cap/samples**.*

## 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
}
```

# 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 / Writes

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

```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

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.

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
```

## 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' }
  ]
}
```

`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.

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

@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.

### Fuzzy Search on SAP HANA Cloud beta

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.

If you run CAP Java in *HEX* optimization mode on SAP HANA Cloud, you can enable fuzzy search in the *application.yaml* and configure the default fuzziness in the range [0.0, 1.0]. The

```yml
cds.sql.hana.search
   fuzzy: true
   fuzzinessThreshold: 0.9
```

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

```cds
entity Books {
  @Search.fuzzinessThreshold: 0.7
  title : 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 '\'.

> **Only Java**
>
> Fuzzy search on SAP HANA Cloud is currently only supported on the CAP Java runtime and requires the `HEX` optimization mode.

## 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`

### 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.

```
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
```

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.

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
```

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 read or updated in a concurrent transaction.

Use *shared* locks if you only need to prevent the entity data to be updated in a concurrent transaction, but don't want to block concurrent read operations.

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

↳ *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.*

> **WARNING**
>
> 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**

`@Core.Computed` , or `@Core.Immutable` (the latter only on UPDATEs).

### `@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/
</Annotations>
```

### `@assert.unique`

Annotate an entity with `@assert.unique.<constraintName>` , specifying one or more element combinations to enforce uniqueness checks on all CREATE and UPDATE operations. For example:

```cds
@assert.unique: {
  locale: [ parent, locale ],
  timeslice: [ parent, validFrom ],
}
entity LocalizedTemporalData {
  key record_ID : UUID; // technical primary key
  parent    : Association to Data;
  locale    : String;
```

```
}
```

This annotation is applicable to entities, which result in tables in SQL databases only.

The value of the annotation is an array of paths referring to elements in the entity. These elements may be of a scalar type, structs, or managed associations. Individual foreign keys or unmanaged associations are not supported.

If structured elements are specified, the unique constraint will contain all columns stemming from it. If the path points to a managed association, the unique constraint will contain all foreign key columns stemming from it.

> **TIP**
>
> You don't need to specify `@assert.unique` constraints for the primary key elements of an entity as these are automatically secured by a SQL `PRIMARY KEY` constraint.

## `@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 .

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

> **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.

### `@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; };
}
```

> Specified ranges are interpreted as closed intervals, that means, the performed checks are $min \leq input \leq max$.

### `@assert.notNull`

Annotate a property with `@assert.notNull: false` to have it ignored during the generic not null check, for example if your persistence fills it automatically.

```cds
entity Foo {
  bar : String not null @assert.notNull: false;
}
```

## 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
```

```
...
```

↳ *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.

- `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;
```

```
  // bound actions & functions
  entity Foo { key ID:Integer } actions {
    function getStock() returns Integer;
    action order (x:Integer) returns Integer;
  }
}
```

↳ *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.

> **Prefer *Unbound* Actions/Functions**
>
> From CDS perspective we recommend **preferring unbound** actions/functions, as these are much more straightforward to implement and invoke.

## 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.

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":1,"to":2}       // unbound action
GET .../sue/Foo(2)/Sue.getStock()     // bound function
POST .../sue/Foo(2)/Sue.order {"x":1} // 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, however, the Node.js runtime also allows calling bound actions/functions without prefixing them with the service name.

**Programmatic** usage via **generic APIs** 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('add',{x:11,to:2})
await srv.send('stock',{id:2})
// bound actions/functions
await srv.send('getStock','Foo',{id:2})
```

```
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.add(11,2)
srv.stock(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.

## 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.

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'` .

> **WARNING**
>
> `@Core.ContentDisposition.Type` is currently only available for the Node.js runtime.

↳ *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:

```
  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.

```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:

Alternatively, you can delete a media data element individually:

```cds
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.

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.env.hana.base64_to_buffer` to `false` .

# Best Practices

## Best Practice: Single-Purposed Services

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
/** 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.

## Best Practice: Late-Cut Microservices

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).

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.

Edit this page

Last updated: 7/16/24, 2:35 PM

Previous page
Domain Modeling

Next page
Consuming Services

# Domain Modeling

Domain Models capture the static, data-related aspects of a problem domain in terms of entity-relationship models. They serve as the basis for *persistence models* deployed to databases as well as for *service definitions*.

Table of Contents


- Introduction
  - Capture Intent – What, not How!
  - Entity-Relationship Modeling
  - Aspect-oriented Modeling
  - Fueling Generic Providers
  - Domain-Driven Design
- Best Practices
  - Keep it Simple, Stupid
  - Separation of Concerns
  - Naming Conventions
- Core Concepts
  - Namespaces
  - Domain Entities
  - Primary Keys
  - Data Types
  - Associations
  - Compositions
- Aspects
  - Authorization

    - Localized Data
- Managed Data
    - *Annotation: @cds.on.insert*
    - *Annotation: @cds.on.update*
    - Aspect managed
- Pseudo Variables


# Introduction

## Capture Intent – *What, not How!*

CDS focuses on *conceptual modelling*: we want to capture intent, not imperative implementations – that is: What, not How. Not only does that keep domain models concise and comprehensible, it also allows us to provide optimized generic implementations.

For example, given an entity definition like that:

```cds
using { cuid, managed } from '@sap/cds/common';
entity Books : cuid, managed {
   title  : localized String;
   descr  : localized String;
   author : Association to Authors;
}
```

In that model we used the pre-defined aspects `cuid` and `managed`, as well as the qualifier `localized` to capture generic aspects. We also used managed associations.

In all these cases, we focus on capturing our intent, while leaving it to generic implementations to provide best-possible implementations.

## Entity-Relationship Modeling

Entity-Relationship Modelling (ERM) is likely the most widely known and applied conceptual modelling technique for data-centric applications. It is also one of the foundations for CDS.

> *"We want to create a bookshop allowing users to browse **Books** and **Authors**, and navigate from Books to Authors and vice versa. Books are classified by **Genre**".*

Using CDS, we would translate that into an initial domain model as follows:

cds

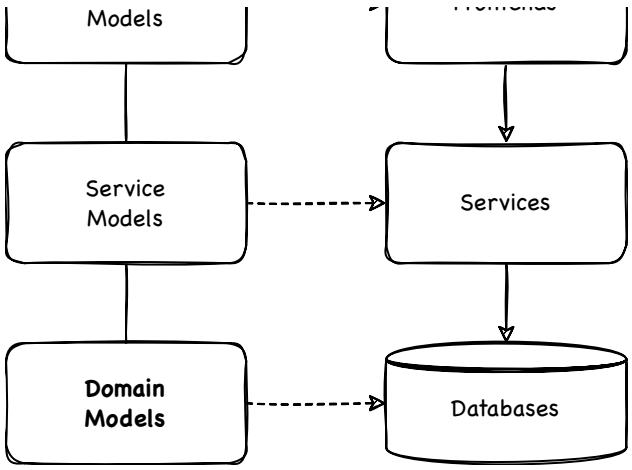
```
using { cuid } from '@sap/cds/common';

entity Books : cuid {
  title  : String;
  descr  : String;
  genre  : Genre;
  author : Association to Authors;
}

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

type Genre : String enum {
  Mystery; Fiction; Drama;
}
```

## Aspect-oriented Modeling

CDS Aspects and Annotations provide powerful means for **separation of concerns**. This greatly helps to keep our core domain model clean, while putting secondary concerns into separate files and model fragments. → Find details in chapter Aspects below.

## Fueling Generic Providers

As depicted in the illustration below, domain models serve as the sources for persistence models, deployed to databases, as well as the underlying model for services acting as API facades to access data.




The more we succeeded in capturing intent over imperative implementations, the more we can provide optimized generic implementations.

### Domain-Driven Design

> **CAP shares these goals and approaches with Domain-driven Design :**
>
> 1. Placing projects' primary **focus on the core domain**
> 2. Close collaboration of **developers** and **domain experts**
> 3. Iteratively refining **domain knowledge**

We use CDS as our ubiquitous modelling language, with CDS Aspects giving us the means to separate core domain aspects from generic aspects. CDS's human-readable nature fosters collaboration of developers and domain experts.

As CDS models are used to fuel generic providers – the database as well as application services – we ensure the models are applied in the implementation. And as coding is minimized we can more easily refine and revise our models, without having to refactor large boilerplate code based.

## Best Practices

### Keep it Simple, Stupid

to understand and work with your models the most, much more than you as their creator. Keep that in mind and understand the task of domain modeling as a service to others.

> **Keep models *concise* and *comprehensible***
>
> As said in the *"Keep it simple, stupid!"* Wikipedia entry: *"... most systems work best if they're kept simple rather than made complicated; therefore, simplicity should be a key goal in design , and unnecessary complexity should be avoided."*

> **Avoid overly abstract models**
>
> Even though domain models should abstract from technical implementations, don't overstress this and balance it with ease of adoption. For example if the vast majority of your clients use relational databases, don't try to overly abstract from that, as that would have all suffer from common denominator syndromes.

### Prefer Flat Models

While CDS provides great support, you should always think twice before using structured types. Some technologies you or your customers use might not integrate with those out of the box. Moreover, flat structures are easier to understand and consume.

**Good:**

cds
```
entity Contacts {
  isCompany : Boolean;
  company   : String;
  title     : String;
  firstname : String;
  lastname  : String;
}
```

**Bad:**

cds
```
entity Contacts {
  isCompany   : Boolean;
  companyData : CompanyDetails;
  personData  : PersonDetails;
}
type CompanyDetails {
  name : String;
```

```
type PersonDetails {
  titles : AcademicTitles;
  name   : PersonName;
}
type PersonName : {
  first  : String;
  last   : String;
}
type AcademicTitles : {
  primary   : String;
  secondary : String;
}
```

## Separation of Concerns

As highlighted with a few samples in the chapter above, always strive to keep your core domain model clean, concise and comprehensible.

CDS Aspects help you to do so, by decomposing models and definitions into separate files with potentially different life cycles, contributed by different *people*.

We strongly recommend to make use of that as much as possible.

## Naming Conventions

We recommend adopting the following simple naming conventions as commonly used in many communities, for example, Java, JavaScript, C, SQL, etc.

To easily distinguish type / entity names from elements names we recommend to...

> **Capitalize *Type / Entity* Names**
>
> - Start ***entity*** and ***type*** names with capital letters – for example, `Authors`
> - Start ***elements*** with a lowercase letter – for example, `name`

As entities represent not only data types, but also data sets, from which we can read from, we recommend following common SQL convention:

- Use **plural** form for ***entities*** – for example, `Authors`
- Use **singular** form for ***types*** – for example, `Genre`

In general always prefer conciseness, comprehensibility and readability, and avoid overly lengthy names, probably dictated by overly strict systematics:

**Prefer *Concise* Names**

- Don't repeat contexts → for example `Authors.name` instead of `Authors.authorName`
- Prefer one-word names → for example `address` instead of `addressInformation`
- Use `ID` for technical primary keys → see also Use Canonic Primary Keys

# Core Concepts

## Namespaces

You can use namespaces to get to unique names without bloating your code with fully qualified names. For example:

```cds
namespace foo.bar;
entity Boo {}
entity Moo : Boo {}
```

... is equivalent to:

```cds
entity foo.bar.Boo {}
entity foo.bar.Moo : foo.bar.Boo {}
```

Note:

in a file. Beyond this there's nothing special about them.

- **Namespaces are optional** – use namespaces if your models might be reused in other projects; otherwise, you can go without namespaces.
- The **reverse domain name** approach works well for choosing namespaces.

> **WARNING**
>
> Avoid short-lived ingredients in namespaces, or names in general, such as your current organization's name, or project code names.

## Domain Entities

Entities represent a domain's data. When translated to persistence models, especially relational ones, entities become tables.

Entity definitions essentially declare structured types with named and typed elements, plus the primary key elements used to identify entries.

```cds
entity name {
   key element1 : Type;
       element2 : Type;
   ...
}
```

↳ *Learn more about entity definitions*

### Views / Projections

Borrowing powerful view building from SQL, we can declare entities as (denormalized) views on other entities:

```cds
entity ProjectedEntity as select from BaseEntity {
   element1, element2 as name, /*...*/
};
```

↳ *Learn more about views and projections*

## Primary Keys

cds

```
entity Books {
  key ID : UUID;
  ...
}
```

**Do:**

- Prefer ***simple***, ***technical*** primary keys
- Prefer ***canonic*** primary keys
- Prefer ***UUIDs*** for primary keys

**Don't:**

- Don't use binary data as keys!
- Don't interpret UUIDs!

### Prefer Simple, Technical Keys

While you can use arbitrary combinations of fields as primary keys, keep in mind that primary keys are frequently used in joins all over the place. And the more fields there are to compare for a join the more you'll suffer from poor performance. So prefer primary keys consisting of single fields only.

Moreover, primary keys should be immutable, that means once assigned on creation of a record they should not change subsequently, as that would break references you might have handed out. Think of them as a fingerprint of a record.

### Prefer Canonic Keys

We recommend using canonically named and typed primary keys, as promoted by aspect `cuid` from @sap/cds/common.

cds

```
// @sap/cds/common
aspect cuid { key ID : UUID }
```

cds

```
using { cuid } from '@sap/cds/common';
entity Books : cuid { ... }
entity Authors : cuid { ... }
```

addressing instances across different types of entities.

### Prefer UUIDs for Keys

While UUIDs certainly come with an overhead and a performance penalty when looking at single databases, they have several advantages when we consider the total bill. So, you can avoid the evil of premature optimization by at least considering these points:

- **UUIDs are universal** – that means that they're unique across every system in the world, while sequences are only unique in the source system's boundaries. Whenever you want to exchange data with other systems you'd anyways add something to make your records 'universally' addressable.
- **UUIDs allow distributed seeds** – for example, in clients. In contrast, database sequences or other sequential generators always need a central service, for example, a single database instance and schema. This becomes even more a problem in distributed landscape topologies.
- **Database sequences are hard to guess** – assume that you want to insert a *SalesOrder* with three *SalesOrderItems* in one transaction. INSERT *SalesOrder* will automatically get a new ID from the sequence. How would you get this new ID in order to use it for the foreign keys in subsequent INSERTs of the *SalesOrderItems*?
- **Auto-filled primary keys** – primary key elements with type UUID are automatically filled by generic service providers in Java and Node.js upon INSERT.

> **Prefer UUIDs for Keys**
>
> Use DB sequences only if you really deal with high data volumes. Otherwise, prefer UUIDs.

You can also have semantic primary keys such as order numbers constructed by customer name+date, etc. And if so, they usually range between UUIDs and DB sequences with respect to the pros and cons listed above.

### Don't Interpret UUIDs!

It is an unfortunate anti pattern to validate UUIDs, such as for compliance to RFC 4122. This not only means useless processing, it also impedes integration with existing data sources. For example, ABAP's **GUID_32s** are uppercase without hyphens.

**UUIDs are unique opaque values!** – The only assumption required and allowed is that UUIDs are unique so that they can be used for lookups and compared by equality – nothing else! It's the task of the UUID generator to ensure uniqueness, not the task of subsequent processors!

representations such as `java.lang.UUID`, only to render them back to strings in responses to HTTP requests, is useless overhead.

> **WARNING**
>
> - Avoid unnecessary assumptions, for example, about uppercase or lowercase
> - Avoid useless conversions, for example, from strings to binary and back
> - Avoid useless validations of UUID formats, for example, about hyphens

↳ *See also: Mapping UUIDs to OData*

↳ *See also: Mapping UUIDs to SQL*

## Data Types

### Standard Built-in Types

CDS comes with a small set of built-in types:

- `UUID`,
- `Boolean`,
- `Date`, `Time`, `DateTime`, `Timestamp`
- `Integer`, `UInt8`, `Int16`, `Int32`, `Int64`
- `Double`, `Decimal`
- `String`, `LargeString`
- `Binary`, `LargeBinary`

↳ *See list of **Built-in Types** in the CDS reference docs*

### Common Reuse Types

In addition, a set of common reuse types and aspects is provided with package `@sap/cds/common`, such as:

- Types `Country`, `Currency`, `Language` with corresponding value list entities
- Aspects `cuid`, `managed`, `temporal`

For example, usage is as simple as this:

```
entity Addresses : managed { //> using reuse aspect
  street  : String;
  town    : String;
  country : Country; //> using reuse type
}
```

↳ *Learn more about reuse types provided by `@sap/cds/common`.*

> **Use common reuse types and aspects...**
>
> ... to keep models concise, and benefitting from improved interoperability, proven best practices, and out-of-the-box support through generic implementations in CAP runtimes.

### Custom-defined Types

Declare custom-defined types to increase semantic expressiveness of your models, or to share details and annotations as follows:

```cds
type User : String; //> merely for increasing expressiveness
type Genre : String enum { Mystery; Fiction; ... }
type DayOfWeek : Number @assert.range:[1,7];
```

### Use Custom Types Reasonably

Avoid overly excessive use of custom-defined types. They're valuable when you have a decent **reuse ratio**. Without reuse, your models just become harder to read and understand, as one always has to look up respective type definitions, as in the following example:

```cds
using { sap.capire.bookshop.types } from './types';
namespace sap.capire.bookshop;
entity Books {
  key ID : types.BookID;
  name : types.BookName;
  descr : types.BookDescr;
  ...
}
```

```cds
// types.cds
namespace sap.capire.bookshop.types;
type BookID : UUID;
```

```
type BookDescr : String;
```

## Associations

Use *Associations* to capture relationships between entities.

cds
```
entity Books { ...
  author : Association to Authors; //> to one
}
entity Authors { ...
  books : Association to many Books on books.author = $self;
}
```

↳ *Learn more about Associations in the CDS Language Reference*

### Managed :1 Associations

The association `Books:author` in the sample above is a so-called *managed* association, with foreign key columns and on conditions added automatically behind the scenes.

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

In contrast to that we could also use *unmanaged* associations with all foreign keys and on conditions specified manually:

cds
```
entity Books { ...
  author : Association to Authors on author.ID = author_ID;
  author_ID : type of Authors:ID;
}
```

> Note: To-many associations are unmanaged by nature as we always have to specify an on condition. Reason for that is that backlink associations or foreign keys cannot be guessed reliably.

**Prefer managed associations**

*Associations* for to-one associations.

### To-Many Associations

Simply add the *many* qualifier keyword to indicate a to-many cardinality:

```cds
entity Authors { ...
  books : Association to many Books;
}
```

If your models are meant to target APIs, this is all that is required. When targeting databases though, we need to add an *on* condition, like so:

```cds
entity Authors { ...
  books : Association to many Books on books.author = $self;
}
```

> The *on* condition can either compare a backlink association to *$self* , or a backlink foreign key to the own primary key, for example *books.author.ID = ID* .

### Many-to-Many Associations

CDS currently doesn't provide dedicated support for *many-to-many* associations. Unless we add some, you have to resolve *many-to-many* associations into two *one-to-many* associations using a link entity to connect both. For example:

```cds
entity Projects { ...
  members : Composition of many Members on members.project = $self;
}
entity Users { ...
  projects : Composition of many Members on projects.user = $self;
}
entity Members: cuid { // link table
  project : Association to Projects;
  user : Association to Users;
}
```

We can use *Compositions of Aspects* to reduce noise a bit:

```
  members : Composition of many { key user : Association to Users };
}
entity Users { ...
  projects : Composition of many Projects.members on projects.member = $self;
}
```

Behind the scenes the equivalent of the model above would be generated, with the link table called *`Projects.members`* and the backlink association to *`Projects`* in there called *`up_`*. Consider that for SAP Fiori elements 'project' and 'user' shall not be keys, even if their combination is unique, because as keys those fields can't be edited on the UI. In this case a different key is required, for example a UUID, and the unique constraint for *`project`* and *`user`* can be expressed via *`@assert.unique`*.

## Compositions

Compositions represent contained-in relationships. CAP runtimes provide these special treatments to Compositions out of the box:

- **Deep Insert / Update** automatically filling in document structures
- **Cascaded Delete** is when deleting Composition roots
- **Composition** targets are **auto-exposed** in service interfaces

### Modeling Document Structures

Compositions are used to model document structures. For example, in the following definition of *`Orders`*, the *`Orders:Items`* composition refers to the *`OrderItems`* entity, with the entries of the latter being fully dependent objects of *`Orders`*.

cds

```
entity Orders { ...
  Items : Composition of many OrderItems on Items.parent = $self;
}
entity OrderItems { // to be accessed through Orders only
  key parent : Association to Orders;
  key pos    : Integer;
  quantity   : Integer;
}
```

↳ *Learn more about Compositions in the CDS Language Reference*

We can use anonymous inline aspects to rewrite the above with less noise as follows:

cds
```
entity Orders { ...
  Items : Composition of many {
    key pos  : Integer;
    quantity : Integer;
  };
}
```

↳ *Learn more about Compositions of Aspects in the CDS Language Reference*

Behind the scenes this will add an entity named `Orders.Items` with a backlink association named `up_`, so effectively generating the same model as above. You can annotate the inline composition with UI annotations as follows:

cds
```
annotate Orders.Items with @(
   UI.LineItem : [
      {Value: pos},
      {Value: quantity},
   ],
);
```

## Aspects

CDS's Aspects provide powerful mechanisms to separate concerns. It allows decomposing models and definitions into separate files with potentially different life cycles, contributed by different *people*.

The basic mechanism use the `extend` or `annotate` directives to add secondary aspects to a core domain entity like so:

cds
```
extend Books with {
   someAdditionalField : String;
}
```

```
  title @some.field.level.annotations;
};
```

Variants of this allow declaring and applying **named aspects** like so:

```cds
aspect NamedAspect { someAdditionalField : String }
extend Books with NamedAspect;
```

We can also apply named aspects as **includes** in an inheritance-like syntax:

```cds
entity Books : NamedAspect { ... }
```

↳ *Learn more about Aspects in the CDS Language Reference*

> **TIP**
>
> Consumers always see the merged effective models, with the separation into aspects fully transparent to them.

## Authorization

CAP supports out-of-the-box authorization by annotating services and entities with *`@requires`* and *`@restrict`* annotations like that:

```cds
entity Books @(restrict: [
  { grant: 'READ',   to: 'authenticated-user' },
  { grant: 'CREATE', to: 'content-maintainer' },
  { grant: 'UPDATE', to: 'content-maintainer' },
  { grant: 'DELETE', to: 'admin' },
]) {
  ...
}
```

To avoid polluting our core domain model with the generic aspect of authorization, we can use aspects to separate concerns, putting the authorization annotations into a separate file, maintained by security experts like so:

```cds
// core domain model in schema.cds
entity Books { ... }
```

cds
```
// authorization model
using { Books, Authors } from './schema.cds';

annotate Books with @restrict: [
  { grant: 'READ',   to: 'authenticated-user' },
  { grant: 'CREATE', to: 'content-maintainer' },
  { grant: 'UPDATE', to: 'content-maintainer' },
  { grant: 'DELETE', to: 'admin' },
];

annotate Authors with @restrict: [
  ...
];
```

## Fiori Annotations

Similarly to authorization annotations we would frequently add annotations which are related to UIs, starting with *`@title`* s used for field or column labels in UIs, or specific Fiori annotations in *`@UI`* , *`@Common`* , etc. vocabularies.

Also here we strongly recommend to keep the core domain models clean of that, but put such annotation into respective frontend models:

cds
```
// core domain model in db/schema.cds
entity Books : cuid { ... }
entity Authors : cuid { ... }
```

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

annotate my.Books with {
  ID     @title: '{i18n>ID}';
  title  @title: '{i18n>Title}';
  genre  @title: '{i18n>Genre}'   @Common: { Text: genre.name, TextArrangemen
  author @title: '{i18n>Author}'  @Common: { Text: author.name, TextArrangeme
  price  @title: '{i18n>Price}'   @Measures.ISOCurrency : currency_code;
```

```
}
```

cds

```
// Specific UI Annotations for Fiori Object & List Pages
using { sap.capire.bookshop as my } from '../db/schema';

annotate my.Books with @(
  Common.SemanticKey : [ID],
  UI: {
    Identification  : [{ Value: title }],
    SelectionFields : [ ID, author_ID, price, currency_code ],
    LineItem        : [
      { Value: ID, Label: '{i18n>Title}' },
      { Value: author.ID, Label: '{i18n>Author}' },
      { Value: genre.name },
      { Value: stock },
      { Value: price },
      { Value: currency.symbol },
    ]
  }
) {
  ID @Common: {
    SemanticObject : 'Books',
    Text: title, TextArrangement : #TextOnly
  };
  author @ValueList.entity: 'Authors';
};
```

## Localized Data

Business applications frequently need localized data, for example to display books titles and descriptions in the user's preferred language. With CDS we simply use the *`localized`* qualifier to tag respective text fields in your as follows.

**Do:**

cds

```
entity Books { ...
  title : localized String;
```

```
}
```

**Don't:**

In contrast to that, this is what you would have to do without CAP's *localized* support:

cds

```
entity Books {
  key ID : UUID;
   title : String;
   descr : String;
   texts : Composition of many Books.texts on texts.book = $self;
   ...
}

entity Books.texts {
  key locale : Locale;
  key ID : UUID;
  title  : String;
  descr  : String;
}
```

Essentially, this is also what CAP generates behind the scenes, plus many more things to ease working with localized data and serving it out of the box.

> **TIP**
>
> By generating *.texts* entities and associations behind the scenes, CAP's **out-of-the-box support** for *localized* data avoids polluting your models with doubled numbers of entities, and detrimental effects on comprehensibility.

↳ *Learn more in the **Localized Data** guide.*

## Managed Data

### *Annotation: @cds.on.insert*

Use the annotations `@cds.on.insert` and `@cds.on.update` to signify elements to be auto-filled by the generic handlers upon insert and update. For example, you could add fields to track who created and updated data records and when:

cds
```
entity Foo { //...
   createdAt  : Timestamp @cds.on.insert: $now;
   createdBy  : User      @cds.on.insert: $user;
   modifiedAt : Timestamp @cds.on.insert: $now  @cds.on.update: $now;
   modifiedBy : User      @cds.on.insert: $user @cds.on.update: $user;
}
```

↳ *Learn more about pseudo variables `$now` and `$user` below.*

These **rules** apply:

- Data *cannot* be filled in from external clients → payloads are cleansed
- Data *can* be filled in from custom handlers or from `.csv` files

▶ *Note the differences to defaults...*

> **In Essence:**
>
> Managed data fields are filled in automatically and are write-protected for externa clients.

> **Limitations**
>
> In case of `UPSERT` operations, the handlers for `@cds.on.update` are executed, but not the ones for `@cds.on.insert` .

## Aspect *managed*

You can also use the pre-defined aspect `managed` from @sap/cds/common to get the very same as by the definition above:

cds
```
using { managed } from '@sap/cds/common';
entity Foo : managed { /*...*/ }
```

↳ *Learn more about `@sap/cds/common`*

## Pseudo Variables

The pseudo variables used in the annotations above are resolved as follows:

- `$now` is replaced by the current server time (in UTC)
- `$user` is the current user's ID as obtained from the authentication middleware
- `$user.<attr>` is replaced by the value of the respective attribute of the current user
- `$uuid` is replaced by a version 4 UUID

↳ *Learn more about **Authentication** in Node.js.*
↳ *Learn more about **Authentication** in Java.*

Edit this page

Last updated: 7/16/24, 2:35 PM

Previous page
Cookbook

Next page
Providing Services