Using Databases

This guide provides instructions on how to use databases with CAP applications. Out of the box-support is provided for SAP HANA, SQLite, H2 (Java only), and PostgreSQL.

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

Table of Contents

Setup & Configuration

Migrating to the `@cap-js/` Database Services?

With CDS 8, the `@cap-js` database services for SQLite, PostgreSQL, and SAP HANA are generally available. It's highly recommended to migrate. You can find instructions in the migration guide. Although the guide is written in the context of the SQLite Service, the same hints apply to PostgreSQL and SAP HANA.

Adding Database Packages

Following are cds-plugin packages for CAP Node.js runtime that support the respective databases:

Database	Package	Remarks
SAP HANA Cloud	`@cap-js/hana`	recommended for production
SQLite	`@cap-js/sqlite`	recommended for development

Database	Package	Remarks
PostgreSQL	*@cap-js/postgres*	maintained by community + CAP team

> Follow the preceding links to find specific information for each.

In general, all you need to do is to install one of the database packages, as follows:

Using SQLite for development:

```sh
npm add @cap-js/sqlite -D
```

Using SAP HANA for production:

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

▶ *Prefer* *`cds add hana`* *...*

Auto-Wired Configuration

The afore-mentioned packages use *`cds-plugin`* techniques to automatically configure the primary database with *`cds.env`*. For example, if you added SQLite and SAP HANA, this effectively results in this auto-wired configuration:

```json
{"cds":{
  "requires": {
    "db": {
      "[development]": { "kind": "sqlite", "impl": "@cap-js/sqlite", "credent
      "[production]": { "kind": "hana", "impl": "@cap-js/hana", "deploy-forma
    }
  }
}}
```

▶ *In contrast to pre-CDS 7 setups this means...*

Custom Configuration

The auto-wired configuration uses configuration presets, which are automatically enabled via `cds-plugin` techniques. You can always use the basic configuration and override individual properties to create a different setup:

1. Install a database driver package, for example:

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

   > Add option `-D` if you want this for development only.

2. Configure the primary database as a required service through `cds.requires.db`, for example:

   ```json
   {"cds":{
     "requires": {
       "db": {
         "kind": "sqlite",
         "impl": "@cap-js/sqlite",
         "credentials": {
           "url": "db.sqlite"
         }
       }
     }
   }}
   ```

The config options are as follows:

- `kind` — a name of a preset, like `sql`, `sqlite`, `postgres`, or `hana`
- `impl` — the module name of a CAP database service implementation
- `credentials` — an object with db-specific configurations, most commonly `url`

Don't configure credentials

Credentials like `username` and `password` should **not** be added here but provided through service bindings, for example, via `cds bind`.

Use `cds env` to inspect effective configuration

For example, running this command:

```sh
cds env cds.requires.db
```

→ prints:

```sh
{
  kind: 'sqlite',
  impl: '@cap-js/sqlite',
  credentials: { url: 'db.sqlite' }
}
```

Providing Initial Data

You can use CSV files to fill your database with initial data - see Location of CSV Files.

For example, in our *capire/bookshop* application, we do so for *Books*, *Authors*, and *Genres* as follows:

```zsh
bookshop/
├─ db/
│ ├─ data/ # place your .csv files here
│ │ ├─ sap.capire.bookshop-Authors.csv
│ │ ├─ sap.capire.bookshop-Books.csv
│ │ ├─ sap.capire.bookshop-Books.texts.csv
│ │ └─ sap.capire.bookshop-Genres.csv
│ └─ schema.cds
└─ ...
```

The **filenames** are expected to match fully qualified names of respective entity definitions in your CDS models, optionally using a dash `-` instead of a dot `.` for cosmetic reasons.

Using *.csv* Files

The **content** of these files is standard CSV content with the column titles corresponding to declared element names, like for `Books`:

db/data/sap.capire.bookshop-Books.csv

csvc
```
ID,title,author_ID,stock
201,Wuthering Heights,101,12
207,Jane Eyre,107,11
251,The Raven,150,333
252,Eleonora,150,555
271,Catweazle,170,22
```

> Note: `author_ID` is the generated foreign key for the managed Association `author` → learn more about that in the Generating SQL DDL section.

If your content contains ...

- commas or line breaks → enclose it in double quotes `"..."`
- double quotes → escape them with doubled double quotes: `""...""`

csvc
```
ID,title,descr
252,Eleonora,"""Eleonora"" is a short story by Edgar Allan Poe, first publish
```

DANGER

On SAP HANA, only use CSV files for *configuration data* that can't be changed by application users. → See CSV data gets overridden in the SAP HANA guide for details.

Use *cds add data*

Run this to generate an initial set of empty `.csv` files with header lines based on your CDS model:

sh
```
cds add data
```

Location of CSV Files

CSV files can be found in the folders *db/data* and *test/data*, as well as in any *data* folder next to your CDS model files. When you use `cds watch` or `cds deploy`, CSV files are loaded by default from *test/data*. However, when preparing for production deployments using `cds build`, CSV files from *test/data* are not loaded.

▶ *Adding initial data next to your data model*

▶ *On SAP HANA ...*

Quite frequently you need to distinguish between sample data and real initial data. CAP supports this by allowing you to provide initial data in two places:

Location	Deployed...	Purpose
db/data	always	initial data for configurations, code lists, and similar
test/data	if not in production	sample data for tests and demos

Querying at Runtime

Most queries to databases are constructed and executed from generic event handlers of CRUD requests, so quite frequently there's nothing to do. The following is for the remaining cases where you have to provide custom logic, and as part of it execute database queries.

DB-Agnostic Queries

At runtime, we usually construct and execute queries using cds.ql APIs in a database-agnostic way. For example, queries like this are supported for all databases:

```js
SELECT.from (Authors, a => {
  a.ID, a.name, a.books (b => {
    b.ID, b.title
  })
})
.where ({name:{like:'A%'}})
.orderBy ('name')
```

Standard Operators

The database services guarantee the identical behavior of these operators:

- `==` , `=` — with `=` null being translated to `is null`
- `!=` , `<>` — with `!=` translated to `IS NOT` in SQLite, or to `IS DISTINCT FROM` in standard SQL, or to an equivalent polyfill in SAP HANA
- `<` , `>` , `<=` , `>=` , `IN` , `LIKE` — are supported as is in standard SQL

In particular, the translation of `!=` to `IS NOT` in SQLite — or to `IS DISTINCT FROM` in standard SQL, or to an equivalent polyfill in SAP HANA — greatly improves the portability of your code.

Session Variables

The API shown after this, which includes the function `session_context()` and specific pseudo variable names, is supported by **all** new database services, that is, *SQLite*, *PostgreSQL* and *SAP HANA*. This allows you to write the respective code once and run it on all these databases:

```sql
SELECT session_context('$user.id')
SELECT session_context('$user.locale')
SELECT session_context('$valid.from')
SELECT session_context('$valid.to')
```

Among other things, this allows us to get rid of static helper views for localized data like `localized_de_sap_capire_Books` .

Native DB Queries

If required you can also use native database features by executing native SQL queries:

```js
cds.db.run (`SELECT from sqlite_schema where name like ?`, name)
```

Reading `LargeBinary` / BLOB

Formerly, `LargeBinary` elements (or BLOBs) were always returned as any other data type. Now, they're skipped from `SELECT *` queries. Yet, you can still enforce reading

BLOBs by explicitly selecting them. Then the BLOB properties are returned as readable streams.

```js
- SELECT.from(Books)          //> [{ ID, title, ..., image1, image2 }]
  SELECT.from(Books)          //> [{ ID, title, ... }]
- SELECT(['image1', 'image2']).from(Books) //> [{ image1, image2 }]
  SELECT(['image1', 'image2']).from(Books) //> [{ image1: Readable, image2: Rea
```

↳ *Read more about custom streaming in Node.js.*

Generating DDL Files

When you run your server with *`cds watch`* during development, an in-memory database is bootstrapped automatically, with SQL DDL statements generated based on your CDS models.

You can also do this manually with the CLI command *`cds compile --to <dialect>`* .

Using *`cds compile`*

For example, given these CDS models (derived from *capire/bookshop*):

db/schema.cds

```cds
using { Currency } from '@sap/cds/common';
namespace sap.capire.bookshop;

entity Books {
  key ID : UUID;
  title  : localized String;
  descr  : localized String;
  author : Association to Authors;
  price  : {
    amount   : Decimal;
    currency : Currency;
  }
}

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

srv/cat-service.cds

```cds
using { sap.capire.bookshop as my } from '../db/schema';
service CatalogService {
  entity ListOfBooks as projection on Books {
    *, author.name as author
  }
}
```

Generate an SQL DDL script by running this in the root directory containing both *.cds* files:

```sh
cds compile srv/cat-service --to sql --dialect sqlite > schema.sql
```

Output:

schema.sql

```sql
CREATE TABLE sap_capire_bookshop_Books (
  ID NVARCHAR(36) NOT NULL,
  title NVARCHAR(5000),
  descr NVARCHAR(5000),
  author_ID NVARCHAR(36),
  price_amount DECIMAL,
  price_currency_code NVARCHAR(3),
  PRIMARY KEY(ID)
);

CREATE TABLE sap_capire_bookshop_Authors (
  ID NVARCHAR(36) NOT NULL,
  name NVARCHAR(5000),
  PRIMARY KEY(ID)
);
```

```sql
CREATE TABLE sap_common_Currencies (
  name NVARCHAR(255),
  descr NVARCHAR(1000),
  code NVARCHAR(3) NOT NULL,
  symbol NVARCHAR(5),
  minorUnit SMALLINT,
  PRIMARY KEY(code)
);

CREATE TABLE sap_capire_bookshop_Books_texts (
  locale NVARCHAR(14) NOT NULL,
  ID NVARCHAR(36) NOT NULL,
  title NVARCHAR(5000),
  descr NVARCHAR(5000),
  PRIMARY KEY(locale, ID)
);

CREATE VIEW CatalogService_ListOfBooks AS SELECT
  Books.ID,
  Books.title,
  Books.descr,
  author.name AS author,
  Books.price_amount,
  Books.price_currency_code
FROM sap_capire_bookshop_Books AS Books
LEFT JOIN sap_capire_bookshop_Authors AS author
ON Books.author_ID = author.ID;

--- some more technical views skipped ...
```

TIP

Use the specific SQL dialect (*`hana`* , *`sqlite`* , *`h2`* , *`postgres`*) with *`cds compile --to sql --dialect <dialect>`* to get DDL that matches the target database.

Rules for Generated DDL

A few observations on the generated SQL DDL output:

1. **Tables / Views** — Declared entities become tables, projected entities become views.

2. **Type Mapping** — CDS types are mapped to database-specific SQL types.
3. **Slugified FQNs** — Dots in fully qualified CDS names become underscores in SQL names.
4. **Flattened Structs** — Structured elements like `Books:price` are flattened with underscores.
5. **Generated Foreign Keys** — For managed to-one Associations, foreign key columns are created. For example, this applies to `Books:author`.

In addition, you can use the following annotations to fine-tune generated SQL.

@cds.persistence.skip

Add `@cds.persistence.skip` to an entity to indicate that this entity should be skipped from generated DDL scripts, and also no SQL views to be generated on top of it:

```cds
@cds.persistence.skip
entity Foo {...}                  //> No SQL table will be generated
entity Bar as select from Foo;    //> No SQL view will be generated
```

@cds.persistence.exists

Add `@cds.persistence.exists` to an entity to indicate that this entity should be skipped from generated DDL scripts. In contrast to `@cds.persistence.skip` a database relation is expected to exist, so we can generate SQL views on top.

```cds
@cds.persistence.exists
entity Foo {...}                  //> No SQL table will be generated
entity Bar as select from Foo;    //> The SQL view will be generated
```

▶ *On SAP HANA ...*

@cds.persistence.table

Annotate an entity with `@cds.persistence.table` to create a table with the effective signature of the view definition instead of an SQL view.

```cds
@cds.persistence.table
entity Foo as projection on Bar {...}
```

> All parts of the view definition not relevant for the signature (like `where`, `group by`, ...) are ignored.

Use case for this annotation: Use projections on imported APIs as replica cache tables.

@sql.prepend / append

Use `@sql.prepend` and `@sql.append` to add native SQL clauses to before or after generated SQL output of CDS entities or elements.

Example:

~~~cds
@sql.append: ```sql
  GROUP TYPE foo
  GROUP SUBTYPE bar
```
entity E { ...,
  @sql.append: 'FUZZY SEARCH INDEX ON'
  text: String(100);
}

@sql.append: 'WITH DDL ONLY'
entity V as select from E { ... };
~~~

Output:

```sql
CREATE TABLE E ( ...,
  text NVARCHAR(100) FUZZY SEARCH INDEX ON
) GROUP TYPE foo
GROUP SUBTYPE bar;

CREATE VIEW V AS SELECT ... FROM E WITH DDL ONLY;
```

The following rules apply:

- The value of the annotation must be a string literal.

- The compiler doesn't check or process the provided SQL snippets in any way. You're responsible to ensure that the resulting statement is valid and doesn't negatively impact your database or your application. We don't provide support for problems caused by using this feature.
- If you refer to a column name in the annotation, you need to take care of a potential name mapping yourself, for example, for structured elements.
- Annotation `@sql.prepend` is only supported for entities translating to tables. It can't be used with views or with elements.
- For SAP HANA tables, there's an implicit `@sql.prepend: 'COLUMN'` that is overwritten by an explicitly provided `@sql.prepend`.
- Both `@sql.prepend` and `@sql.append` are disallowed in SaaS extension projects.

If you use native database clauses in combination with `@cds.persistence.journal`, see Schema Evolution Support of Native Database Clauses.

Creating a Row Table on SAP HANA

By using `@sql.prepend: 'ROW'`, you can create a row table:

```cds
@sql.prepend: 'ROW'
entity E {
  key id: Integer;
}
```

Run `cds compile - 2 hdbtable` on the previous sample and this is the result:

```sql
ROW TABLE E (
  id INTEGER NOT NULL,
  PRIMARY KEY(id)
)
```

↳ *Learn more about Columnar and Row-Based Data Storage*

Reserved Words

The CDS compiler and CAP runtimes provide smart quoting for reserved words in SQLite and in SAP HANA so that they can still be used in most situations. But in general reserved words can't be used as identifiers. The list of reserved words varies per database.

Find here a collection of resources on selected databases and their reference documentation:

- SAP HANA SQL Reference Guide for SAP HANA Platform (Cloud Version)
- SAP HANA SQL Reference Guide for SAP HANA Cloud
- SQLite Keywords
- H2 Keywords/Reserved Words
- PostgreSQL SQL Key Words

↳ *There are also reserved words related to SAP Fiori.*

Database Constraints

Not Null

You can specify that a column's value must not be `NULL` by adding the `not null` constraint to the element, for example:

```cds
entity Books {
  key ID: Integer;
  title: String not null;
}
```

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;
    validFrom : Date;
    validTo : Date;
  }
```

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.

Foreign Keys

The information about foreign key relations contained in the associations of CDS models can be used to generate foreign key constraints on the database tables. Within CAP, referential consistency is established only at commit. The "deferred" concept for foreign key constraints in SQL databases allows the constraints to be checked and enforced at the time of the COMMIT statement within a transaction rather than immediately when the data is modified, providing more flexibility in maintaining data integrity.

Enable generation of foreign key constraints on the database with:

`cds.features.assert_integrity = db` ☼

> **Database constraints are not supported for H2**
>
> Referential constraints on H2 cannot be defined as "deferred", which is needed for database constraints within CAP.

With that switched on, foreign key constraints are generated for managed to-one associations. For example, given this model:

```cds
entity Books {
  key ID : Integer; ...
```

```
    author : Association to Authors;
  }
  entity Authors {
    key ID : Integer; ...
  }
```

The following *Books_author* constraint would be added to table *Books* :

```sql
CREATE TABLE Authors (
  ID INTEGER NOT NULL,  -- primary key referenced by the constraint
  ...,
  PRIMARY KEY(ID)
);
CREATE TABLE Books (
  ID INTEGER NOT NULL,
  author_ID INTEGER,    -- generated foreign key field
  ...,
  PRIMARY KEY(ID),
  CONSTRAINT Books_author   -- constraint is explicitly named
    FOREIGN KEY(author_ID)  -- link generated foreign key field author_ID ...
    REFERENCES Authors(ID)  -- ... with primary key field ID of table Authors
    ON UPDATE RESTRICT
    ON DELETE RESTRICT
    VALIDATED           -- validate existing entries when constraint is creat
    ENFORCED            -- validate changes by insert/update/delete
    INITIALLY DEFERRED  -- validate only at commit
)
```

No constraints are generated for...

- Unmanaged associations or compositions
- To-many associations or compositions
- Associations annotated with `@assert.integrity: false`
- Associations where the source or target entity is annotated with `@cds.persistence.exists` or `@cds.persistence.skip`

If the association is the backlink of a **composition**, the constraint's delete rule changes to `CASCADE` . That applies, for example, to the `parent` association in here:

```cds
entity Genres {
  key ID   : Integer;
```

```
    parent   : Association to Genres;
    children : Composition of many Genres on children.parent = $self;
  }
```

As a special case, a referential constraint with `delete cascade` is also generated for the text table of a localized entity, although no managed association is present in the `texts` entity.

Add a localized element to entity `Books` from the previous example:

cds
```
entity Books {
  key ID : Integer; ...
  title : localized String;
}
```

The generated text table then is:

sql
```
CREATE TABLE Books_texts (
  locale NVARCHAR(14) NOT NULL,
  ID INTEGER NOT NULL,
  title NVARCHAR(5000),
  PRIMARY KEY(locale, ID),
  CONSTRAINT Books_texts_texts // [!code focus]
    FOREIGN KEY(ID)
    REFERENCES Books(ID)
    ON UPDATE RESTRICT
    ON DELETE CASCADE
    VALIDATED
    ENFORCED
    INITIALLY DEFERRED
)
```

Database constraints aren't intended for checking user input

Instead, they protect the integrity of your data in the database layer against programming errors. If a constraint violation occurs, the error messages coming from the database aren't standardized by the runtimes but presented as-is.

→ Use `@assert.target` for corresponding input validations.

Standard Database Functions

A specified set of standard functions - inspired by OData and SAP HANA - is supported in a **database-agnostic**, hence portable way. The functions are translated to the best-possible database-specific SQL expressions at runtime and also during compilation of your CDL files.

OData standard functions

The *@sap/cds-compiler* and the database services come with out of the box support for common OData functions.

Case Sensitivity

The OData function mappings are case-sensitive and must be written as in the list below.

Assuming you have the following entity definition:

```cds
entity V as select from Books {
  startswith(title, 'Raven') as lowerCase, // mapped to native SQL equivalent
  startsWith(title, 'Raven') as camelCase, // passed as-is
}
```

Then you compile the SAP HANA artifacts:

```
$ cds compile -2 sql --dialect hana
```

This is the result:

```sql
CREATE VIEW V AS SELECT
  (CASE WHEN locate(title, 'Raven') = 1 THEN TRUE ELSE FALSE END) AS lowerCas
  -- the below will most likely fail on SAP HANA
  startsWith(title, 'Raven') AS camelCase
FROM Books;
```

💡 If you want to use a DB native function or a UDF (User-Defined Function) instead of the OData function mappings, you can do that by using a different casing than the OData function names as defined in the list below. For example, `startsWith` instead of `startswith` will be passed as-is to the database.

String Functions

- `concat(x, y, ...)` Concatenates the given strings or numbers `x` , `y` , ...
- `trim(x)` Removes leading and trailing whitespaces from `x` .
- `contains(x, y)` Checks whether `x` contains `y` (case-sensitive).
- `startswith(x, y)` Checks whether `x` starts with `y` (case-sensitive).
- `endswith(x, y)` Checks whether `x` ends with `y` (case-sensitive).
- `matchespattern(x, y)` Checks whether `x` matches the regular expression `y` .
- `indexof(x, y)` [1] Returns the index of the first occurrence of `y` in `x` (case-sensitive).
- `substring(x, i, n?)` [1] Extracts a substring from `x` starting at index `i` (0-based) with an optional length `n` .

Parameter	Positive	Negative	Omitted
`i`	starts at index `i`	starts `i` positions before the end	
`n`	extracts `n` characters	invalid	extracts until the end of the string

- `length(x)` Returns the length of the string `x` .
- `tolower(x)` Converts all characters in `x` to lowercase.
- `toupper(x)` Converts all characters in `x` to uppercase.

> [1] These functions work zero-based. For example, `substring('abcdef', 1, 3)` returns 'bcd'

Numeric Functions

- `ceiling(x)` Rounds the numeric parameter up to the nearest integer.
- `floor(x)` Rounds the numeric parameter down to the nearest integer.

- `round(x)` Rounds the numeric parameter to the nearest integer. The midpoint between two integers is rounded away from zero (e.g., `0.5 → 1` and `-0.5 → -1`).

> **`round` function with more than one argument**
>
> Note that most databases support `round` functions with multiple arguments, the second parameter being the precision. SAP HANA even has a third argument which is the rounding mode. If you provide more than one argument, the `round` function may behave differently depending on the database.

Date and Time Functions

- `year(x)`, `month(x)`, `day(x)`, `hour(x)`, `minute(x)`, `second(x)` Extracts and returns specific date / time parts as integer value from a given `cds.DateTime`, `cds.Date`, or `cds.Time`.
- `time(x)`, `date(x)` Extracts and returns a time or date from a given `cds.DateTime`, `cds.Date`, or `cds.Time`.
- `fractionalseconds(x)` Returns a `Decimal` representing the fractional seconds for a given `cds.Timestamp`.
- `maxdatetime()` Returns the latest possible point in time: `'9999-12-31T23:59:59.999Z'`.
- `mindatetime()` Returns the earliest possible point in time: `'0001-01-01T00:00:00.000Z'`.

Aggregate Functions

- `min(x)`, `max(x)`, `sum(x)`, `average(x)`, `count(x)`, `countdistinct(x)` Standard aggregate functions used to calculate minimum, maximum, sum, average, count, and distinct count of values.

SAP HANA Functions

In addition to the OData standard functions, the `@sap/cds-compiler` and all CAP Node.js database services come with out of the box support for some common SAP HANA functions, to further increase the scope for portable testing:

Upper- and Lowercase are supported

For the SAP HANA functions, both usages are allowed: all-lowercase as given above, as well as all-uppercase.

- `years_between` Computes the number of years between two specified dates. (link)
- `months_between` Computes the number of months between two specified dates. (link)
- `days_between` Computes the number of days between two specified dates. (link)
- `seconds_between` Computes the number of seconds between two specified dates. (link)
- `nano100_between` Computes the time difference between two dates to the precision of 0.1 microseconds. (link)

Special Runtime Functions

In addition to the OData and SAP HANA standard functions, the **CAP runtime** provides special functions that are only available for runtime queries:

- `search(x, y)` Checks whether `y` is contained in any element of `x` (fuzzy matching may apply). See Searching Data for more details.
- `session_context(<var>)` Utilizes standard variable names to maintain session context. Refer to Session Variables for additional information.
- `now()` Returns the current timestamp.

Using Native Features

In general, the CDS 2 SQL compiler doesn't 'understand' SQL functions but translates them to SQL generically as long as they follow the standard call syntax of `function(param1, param2)`. This allows you to use native database functions inside your CDS models.

Example:

```cds
entity BookPreview as select from Books {
  IFNULL (descr, title) as shorttext   //> using HANA function IFNULL
};
```

The *OVER* clause for SQL Window Functions is supported, too:

```cds
entity RankedBooks as select from Books {
  name, author,
  rank() over (partition by author order by price) as rank
};
```

Using Native Functions with Different DBs

In case of conflicts, follow these steps to provide different models for different databases:

1. Add database-specific schema extensions in specific subfolders of `./db` :

   db/sqlite/index.cds   db/hana/index.cds

   ```cds
   using { AdminService } from '..';
   extend projection AdminService.Authors with {
      strftime('%Y',dateOfDeath)-strftime('%Y',dateOfBirth) as age : Integer
   }
   ```

2. Add configuration in specific profiles to your *package.json*, to use these database-specific extensions:

   ```json
   { "cds": { "requires": {
     "db": {
      "kind": "sql",
      "[development]": { "model": "db/sqlite" },
      "[production]": { "model": "db/hana" }
    }
   }}}
   ```

CAP samples demonstrate this in @capire/bookstore .

Edit this page

Last updated: 05/12/2025, 10:49

Previous page
Serving UIs

Next page
SQLite