Using SQLite for Development

CAP provides extensive support for SQLite , which allows projects to speed up development by magnitudes at minimized costs. We strongly recommend using this option as much as possible during development and testing.

> **New SQLite Service**
>
> This guide focuses on the new SQLite Service provided through *@cap-js/sqlite* , which has many advantages over the former one, as documented in the *Features* section. To migrate from the old service, find instructions in the *Migration* section.

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

Table of Contents

Setup & Configuration

Run this to use SQLite for development:

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

Auto-Wired Configuration

The *`@cap-js/sqlite`* package uses the *`cds-plugin`* technique to auto-configure your application for using an in-memory SQLite database for development.

You can inspect the effective configuration using *`cds env`* :

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

Output:

```js
{
  impl: '@cap-js/sqlite',
  credentials: { url: ':memory:' },
```

```
    kind: 'sqlite'
  }
```

↳ *See also the general information on installing database packages.*

Deployment

In-Memory Databases

As stated previously, `@cap-js/sqlite` uses an in-memory SQLite database by default. For example, when starting your application with `cds watch`, you can see this in the log output:

log

```
...
[cds] - connect to db > sqlite { url: ':memory:' }
  > init from db/init.js
  > init from db/data/sap.capire.bookshop-Authors.csv
  > init from db/data/sap.capire.bookshop-Books.csv
  > init from db/data/sap.capire.bookshop-Books.texts.csv
  > init from db/data/sap.capire.bookshop-Genres.csv
/> successfully deployed to in-memory database.
...
```

> **TIP**
>
> Using in-memory databases is the most recommended option for test drives and test pipelines.

Persistent Databases

You can also use persistent SQLite databases. Follow these steps to do so:

1. Specify a database filename in your `db` configuration as follows:

package.json

```json
{ "cds": { "requires": {
   "db": {
      "kind": "sqlite",
      "credentials": { "url": "db.sqlite" }
   }
}}}
```

2. Run `cds deploy`:

```sh
cds deploy
```

This will:

1. Create a database file with the given name.
2. Create the tables and views according to your CDS model.
3. Fill in initial data from the provided *.csv* files.

With that in place, the server will use this prepared database instead of bootstrapping an in-memory one upon startup:

```log
...
[cds] - connect to db > sqlite { url: 'db.sqlite' }
...
```

> **Redeploy on changes**
>
> Remember to always redeploy your database whenever you change your models or your data. Just run `cds deploy` again to do so.

Drop-Create Schema

When you redeploy your database, it will always drop-create all tables and views. This is **most suitable for development environments**, where schema changes are very frequent and broad.

Schema Evolution

While drop-create is most appropriate for development, it isn't suitable for database upgrades in production, as all customer data would be lost. To avoid this, `cds deploy` also supports automatic schema evolution, which you can use as follows:

1. Enable automatic schema evolution in your `db` configuration:

package.json

```json
{ "cds": { "requires": {
   "db": {
      "kind": "sqlite",
      "credentials": { "url": "db.sqlite" },
      "schema_evolution": "auto"
   }
}}}
```

2. Run `cds deploy` :

```sh
cds deploy
```

↳ *Learn more about automatic schema evolution in the PostgreSQL guide.*
The information in there is also applicable to SQLite with persistent databases.

Features

The following is an overview of advanced features supported by the new database services.

> These apply to all new database services, including SQLiteService, HANAService, and PostgresService.

Path Expressions & Filters

The new database service provides **full support** for all kinds of path expressions, including infix filters and exists predicates. For example, you can try this out with *@capire/samples* as follows:

```js
// $ cds repl --profile better-sqlite
var { server } = await cds.test('bookshop'), { Books, Authors } = cds.entities
await INSERT.into (Books) .entries ({ title: 'Unwritten Book' })
await INSERT.into (Authors) .entries ({ name: 'Upcoming Author' })
await SELECT `from ${Books} { title as book, author.name as author, genre.name
await SELECT `from ${Authors} { books.title as book, name as author, books.genre
await SELECT `from ${Books} { title as book, author[ID<170].name as author, g
await SELECT `from ${Books} { title as book, author.name as author, genre.name
await SELECT `from ${Books} { title as book, author.name as author, genre.name
await SELECT `from ${Books}:author[name like 'Ed%' or ID=170] { books.title as
await SELECT `from ${Books}:author[150] { books.title as book, name as author
await SELECT `from ${Authors} { ID, name, books { ID, title }}`
await SELECT `from ${Authors} { ID, name, books { ID, title, genre { ID, name
await SELECT `from ${Authors} { ID, name, books.genre { ID, name }}`
await SELECT `from ${Authors} { ID, name, books as some_books { ID, title, genre
await SELECT `from ${Authors} { ID, name, books[genre.ID=11] as dramatic_books
await SELECT `from ${Authors} { ID, name, books.genre[name!='Drama'] as no_drama
await SELECT `from ${Authors} { books.genre.ID }`
await SELECT `from ${Authors} { books.genre }`
await SELECT `from ${Authors} { books.genre.name }`
```

Optimized Expands

The old database service implementation(s) used to translate deep reads, that is, SELECTs with expands, into several database queries and collect the individual results into deep result structures. The new service uses *`json_object`* and other similar functions to instead do that in one single query, with sub selects, which greatly improves performance.

For example:

```sql
SELECT.from(Authors, a => {
  a.ID, a.name, a.books (b => {
    b.title, b.genre (g => {
       g.name
    })
  })
})
```

While this used to require three queries with three roundtrips to the database, now only one query is required.

Localized Queries

With the old implementation, running queries like `SELECT.from(Books)` would always return localized data, without being able to easily read the non-localized data. The new service does only what you asked for, offering new `SELECT.localized` options:

```js
let books = await SELECT.from(Books)       //> non-localized data
let lbooks = await SELECT.localized(Books) //> localized data
```

Usage variants include:

```js
SELECT.localized(Books)
SELECT.from.localized(Books)
SELECT.one.localized(Books)
```

Using Lean Draft

The old implementation was overly polluted with draft handling. But as draft is actually a Fiori UI concept, none of that should show up in database layers. Hence, we eliminated all draft handling from the new database service implementations, and implemented draft in a modular, non-intrusive way — called *'Lean Draft'*. The most important change is that we don't do expensive UNIONs anymore but work with single (cheap) selects.

Consistent Timestamps

Values for elements of type `DateTime` and `Timestamp` are handled in a consistent way across all new database services along these lines:

> ***Timestamps* = `Timestamp` as well as `DateTime`**
>
> When we say *Timestamps*, we mean elements of type `Timestamp` as well as `DateTime`. Although they have different precision levels, they are essentially the same type. `DateTime` elements have seconds precision, while `Timestamp` elements have

> milliseconds precision in SQLite, and microsecond precision in SAP HANA and PostgreSQL.

Writing Timestamps

When writing data using INSERT, UPSERT or UPDATE, you can provide values for `DateTime` and `Timestamp` elements as JavaScript `Date` objects or ISO 8601 Strings. All input is normalized to ensure `DateTime` and `Timestamp` values can be safely compared. In case of SAP HANA and PostgreSQL, they're converted to native types. In case of SQLite, they're stored as ISO 8601 Strings in Zulu timezone as returned by JavaScript's `Date.toISOString()` .

For example:

```js
await INSERT.into(Books).entries([
  { createdAt: new Date },                        //> stored .toISOString()
  { createdAt: '2022-11-11T11:11:11Z' },          //> padded with .000Z
  { createdAt: '2022-11-11T11:11:11.123Z' },      //> stored as is
  { createdAt: '2022-11-11T11:11:11.1234563Z' }, //> truncated to .123Z
  { createdAt: '2022-11-11T11:11:11+02:00' },     //> converted to zulu time
])
```

Reading Timestamps

Timestamps are returned as they're stored in a normalized way, with milliseconds precision, as supported by the JavaScript `Date` object. For example, the entries inserted previously would return the following:

```js
await SELECT('createdAt').from(Books).where({title:null})
```

```js
[
  { createdAt: '2023-08-10T14:24:30.798Z' },
  { createdAt: '2022-11-11T11:11:11.000Z' },
  { createdAt: '2022-11-11T11:11:11.123Z' },
  { createdAt: '2022-11-11T11:11:11.123Z' },
  { createdAt: '2022-11-11T09:11:11.000Z' }
]
```

`DateTime` elements are returned with seconds precision, with all fractional second digits truncated. That is, if the `createdAt` in our examples was a `DateTime` , the previous

query would return this:

```js
[
  { createdAt: '2023-08-10T14:24:30Z' },
  { createdAt: '2022-11-11T11:11:11Z' },
  { createdAt: '2022-11-11T11:11:11Z' },
  { createdAt: '2022-11-11T11:11:11Z' },
  { createdAt: '2022-11-11T09:11:11Z' }
]
```

Comparing DateTimes & Timestamps

You can safely compare DateTimes & Timestamps with each other and with input values. The input values have to be *`Date`* objects or ISO 8601 Strings in Zulu timezone with three fractional digits.

For example, all of these would work:

```js
SELECT.from(Foo).where `someTimestamp = anotherTimestamp`
SELECT.from(Foo).where `someTimestamp = someDateTime`
SELECT.from(Foo).where `someTimestamp = ${new Date}`
SELECT.from(Foo).where `someTimestamp = ${req.timestamp}`
SELECT.from(Foo).where `someTimestamp = ${'2022-11-11T11:11:11.123Z'}`
```

While these would fail, because the input values don't comply to the rules:

```js
SELECT.from(Foo).where `createdAt = ${'2022-11-11T11:11:11+02:00'}` // non-Zu
SELECT.from(Foo).where `createdAt = ${'2022-11-11T11:11:11Z'}` // missing 3-d
```

> This is because we can never reliably infer the types of input to *`where`* clause expressions. Therefore, that input will not receive any normalisation, but be passed down as is as plain string.

> **Always ensure proper input in *`where`* clauses**
>
> Either use strings strictly in *`YYYY-MM-DDThh:mm:ss.fffZ`* format, or *`Date`* objects, as follows:
>
> ```js
> SELECT.from(Foo).where ({ createdAt: '2022-11-11T11:11:11.000Z' })
> SELECT.from(Foo).where ({ createdAt: new Date('2022-11-11T11:11:11Z') })
> ```

The rules regarding Timestamps apply to all comparison operators: `=`, `<`, `>`, `<=`, `>=`.

Improved Performance

The combination of the above-mentioned improvements commonly leads to significant performance improvements. For example, displaying the list page of Travels in capire/xtravels took **>250ms** in the past, and **~15ms** now.

Migration

While we were able to keep all public APIs stable, we had to apply changes and fixes to some **undocumented behaviours and internal APIs** in the new implementation. While not formally breaking changes, you may have used or relied on these undocumented APIs and behaviours. In that case, you can find instructions about how to resolve this in the following sections.

> These apply to all new database services: SQLiteService, HANAService, and PostgresService.

Use Old and New in Parallel

During migration, you may want to occasionally run and test your app with both the new SQLite service and the old one. You can accomplish this as follows:

1. Add the new service with `--no-save`:

   ```sh
   npm add @cap-js/sqlite --no-save
   ```

   > This bypasses the *cds-plugin* mechanism, which works through package dependencies.

2. Run or test your app with the `better-sqlite` profile using one of these options:

   ```sh
   cds watch bookshop --profile better-sqlite
   ```

```sh
CDS_ENV=better-sqlite cds watch bookshop
```

```sh
CDS_ENV=better-sqlite jest --silent
```

3. Run or test your app with the old SQLite service as before:

```sh
cds watch bookshop
```

```sh
jest --silent
```

Avoid UNIONs and JOINs

Many advanced features supported by the new database services, like path expressions or deep expands, rely on the ability to infer queries from CDS models. This task gets extremely complex when adding UNIONs and JOINs to the equation — at least the effort and overhead is hardly matched by generated value. Therefore, we dropped support of UNIONs and JOINs in CQN queries.

For example, this means queries like these are deprecated / not supported any longer:

```js
SELECT.from(Books).join(Authors,...)
```

Mitigations:

1. Use path expressions instead of joins. (The former lack of support for path expressions was the most common reason for having to use joins at all.)
2. Use plain SQL queries like so:

```js
await db.run(`SELECT from ${Books} join ${Authors} ...`)
```

3. Use helper views modeled in CDS, which still supports all complex UNIONs and JOINs, then use this view via `cds.ql`.

Fixed Localized Data

Formerly, when reading data using `cds.ql`, this *always* returned localized data. For example:

```js
SELECT.from(Books)       // always read from localized.Books instead
```

This wasn't only wrong, but also expensive. Localized data is an application layer concept. Database services should return what was asked for, and nothing else. → Use *Localized Queries* if you really want to read localized data from the database:

```js
SELECT.localized(Books)  // reads localized data
SELECT.from(Books)       // reads plain data
```

▶ *No changes to app services behaviour*

Skipped Virtuals

In contrast to their former behaviour, new database services ignore all virtual elements and hence don't add them to result set entries. Selecting only virtual elements in a query leads to an error.

▶ *Reasoning*

For example, given this definition:

```cds
entity Foo {
  foo : Integer;
  virtual bar : Integer;
}
```

The behaviour has changed to:

```js
[dev] cds repl
- > SELECT.from('Foo')         //> [{ foo:1, bar:null }, ...]
  > SELECT.from('Foo')         //> [{ foo:1 }, ...]
  > SELECT('bar').from('Foo')  //> ERROR: no columns to read
```

<> Operator

Before, both `<>` and `!=` were translated to `name <> 'John' OR name is null`.

- The operator `<>` now works as specified in the SQL standard.
- `name != 'John'` is translated as before to `name <> 'John' OR name is null`.

> **WARNING**
>
> This is a breaking change in regard to the previous implementation.

Miscellaneous

- Only `$now` and `$user` are supported as values for `@cds.on.insert/update`.
- Managed fields are automatically filled with `INSERT.entries()`, but not when using `INSERT.columns().values()` or `INSERT.columns().rows()`.
- If the column of a `SELECT` is a path expression without an alias, the field name in the result is the concatenated name using underscores. For example, `SELECT.from(Books).columns('author.name')` results in `author_name`.
- CQNs with subqueries require table aliases to refer to elements of outer queries.
- Table aliases must not contain dots.
- CQNs with an empty columns array now throw an error.
- `*` isn't a column reference. Use `columns: ['*']` instead of `columns: [{ref:'*'}]`.
- Column names in CSVs must map to physical column names:

```csvc
ID;title;author_ID;currency_code // [!code ++]
ID;title;author.ID;currency.code // [!code --]
```

Adopt Lean Draft

As mentioned in Using Lean Draft, we eliminated all draft handling from new database service implementations, and instead implemented draft in a modular, non-intrusive, and optimized way — called *'Lean Draft'*.

When using the new service, the new `cds.fiori.lean_draft` mode is automatically switched on.

More detailed documentation for that is coming.

Finalizing Migration

When you have finished migration, remove the old *sqlite3* driver :

```sh
npm rm sqlite3
```

And activate the new one as cds-plugin:

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

SQLite in Production?

As stated in the beginning, SQLite is mostly intended to speed up development, but is not fit for production. This is not because of limited warranties or lack of support, but rather because of suitability.

A major criterion is this: cloud applications are usually served by server clusters, in which each server is connected to a shared database. SQLite could only be used in such setups with the persistent database file accessed through a network file system. This is rarely available and results in slow performance. Hence, an enterprise client-server database is a more fitting choice for these scenarios.

Having said this, there can indeed be scenarios where SQLite might also be used in production, such as using SQLite as in-memory caches. → Find a detailed list of criteria on the sqlite.org website .

> **WARNING**
>
> SQLite only has limited support for concurrent database access due to its very coarse lock granularity. This makes it badly suited for applications with high concurrency.

Databases H2 (Java)

Was this page helpful?